HIGH-PERFORMANCE DIRECT SOLUTION OF FINITE ELEMENT PROBLEMS ON MULTI-CORE PROCESSORS

A Dissertation
Presented to
The Academic Faculty

by

Murat Efe Guney

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Civil and Environmental Engineering

Georgia Institute of Technology
August 2010

HIGH-PERFORMANCE DIRECT SOLUTION OF FINITE ELEMENT PROBLEMS ON MULTI-CORE PROCESSORS

Approved by:

Dr. Kenneth M. Will, Advisor
School of Civil and Environmental Engineering
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Donald W. White
School of School of Civil and Environmental Engineering
Georgia Institute of Technology

Dr. Ozgur Kurc
Civil Engineering Department
Middle East Technical University, Ankara, Turkey

Dr. Leroy Z. Emkin
School of School of School of Civil and Environmental Engineering
Georgia Institute of Technology

Date Approved: May 3, 2010

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. Kenneth M. Will, for his support and patience. I'm grateful for his continuous guidance and trust. He always encouraged me to explore different fields and made my PhD experience as exciting as it could be. Under his supervision, I've got more than what I've dreamed.

I also would like to thank Dr. Richard Vuduc for his support and guidance. He provided me insights about various topics in high-performance computing. I'm also grateful to him for giving me access to his computer resources.

I also would like to thank Dr. Ozgur Kurc. His previous work provided me insight about parallel computing in structural mechanics. Five years ago, it all started with reading his thesis and browsing his code.

I also would like to thank Dr. Donald White for his support and recommendations. I admire his knowledge and inspiration. I will miss our discussions about structural stability and second-order analysis.

I also would like to thank Dr. Leroy Emkin for taking the time to serve on my qualifications and committee. His comments and questions make me to continuously increase the size of my test problems.

I would like to thank my friend and roommate, Cagri Ozgur, for his never-ending support. He is in half of my fun memories of the past 5 years. I also wish to thank Erinc Atilla for being a true friend for more than 20 years. We owe him a lot for his support and guidance after we came to A-town. I also express my deepest gratitude to Sorian Enriquez for her support and caring. Furthermore, I would like to thank my office mates,

Gwang-Seok Na, Ben Deaton, Jong-Han Lee, Mustafa Can Kara, Jennifer Modugno and Jennifer Dunbeck. Delicious cakes baked by Jennifer's gave me the energy required to finish this dissertation. I also would like to thank all of my colleagues in School of Civil & Environmental Engineering, especially Jie-Eun Hur, Jon Hurff, Andres Sanchez, Gence Genc, Masahiro Kurata, Sibel Kerpicci, Yoon-Duk Kim, Shane Johnson, Towhid Ahammad, Murat Engindeniz, Yavuz Mentes, Ilker Kalkan, Cem Ozan, Juan Jimenez, Akhil Sharma, Robert Moser, Murat Eroz, and Ozan Cem Celik.

Words cannot express my heartfelt gratitude to my beloved parents, Sevtap Guney and Sukru Guney, and my dearest brother, Emre Guney, who reviewed some parts of this dissertation. None of this would have been possible without their unconditional love and support.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

SUMMARY

The solution of linear system of equations is at the core of finite element (FE) analysis software. While engineers have been increasing the size and complexity of their models, the growth in the speed of a single computer processor has slowed. Today, computer manufacturers have increased overall processor performance by increasing the number of processing units in a computer using so-called multi-core processors. A FE analysis solver is needed which takes full advantage of these multi-core processors.

In this study, a direct solution procedure is proposed and developed which exploits the parallelism that exists in current symmetric multiprocessing (SMP) multi-core processors. Several algorithms are proposed and developed to improve the performance of the direct solution of FE problems. A high-performance sparse direct solver is developed which allows experimentation with the newly developed and existing algorithms. The performance of the algorithms is investigated using a large set of FE problems. Furthermore, operation count estimations are developed to further assess various algorithms.

A multifrontal method is adopted for the parallel factorization and triangular solution on SMP multi-core processors. A triangular solution algorithm that is especially efficient for the solution with multiple loading conditions is developed. Furthermore, a new mapping algorithm is designed to find independent factorization tasks that are assigned to the CPU cores in order to minimize the parallel factorization time. As the factorization and triangular solution times are reduced by the use of parallel algorithms, other components of FE analysis such as assembly of the stiffness matrix become a bottleneck for improving the overall performance. An assembled stiffness matrix is not required by the developed solver. Instead, element stiffness matrices and element connectivity information are the inputs. The developed solver never assembles the entire

structural stiffness matrix but assembles frontal matrices on each core. This reduces not only the execution time but also the memory requirement for the assembly.

Finally, an out-of-core version of the solver is developed to reduce the memory requirements for the solution. I/O is performed asynchronously without blocking the thread that makes the I/O request. Asynchronous I/O allows overlapping factorization and triangular solution computations with I/O. The performance of the developed solver is demonstrated on a large number of test problems. A problem with nearly 10 million degree of freedoms is solved on a low price desktop computer using the out-of-core version of the direct solver. Furthermore, the developed solver usually outperforms a commonly used shared memory solver.

CHAPTER 1

INTRODUCTION

1.1 Problem Definition

As computational power continues to increase, the size and complexity of analysis problems also increase. Realistic simulations require complex models and responsive software is desired. Large-scale structural simulations can be performed in a reasonable time using software which takes advantage of the computational potential of modern processors.

The efficient use of modern processors can be challenging due to the sophisticated hardware architectures and existence of multiple processing units. Recently, multi-core processors have been introduced into commodity laptop and desktop computers for higher performance. Today, 6-core and dual quad-core PC's are available, and processor manufacturers are planning to increase the number of cores to meet the ever-increasing demand for performance [1-3]. The additional cores caused a paradigm shift in the programming practice. Parallel algorithms are required to harness the computational power introduced by multi-core and many-core processors. Starting from the components with the largest execution time, software built for single core processors must be redesigned in order to benefit from the emerging computational power introduced by the multi-core and many-core processors.

The solution of linear system of equations is the most computationally intensive component of finite element analysis (FEA) software. A sparse direct solver optimized for the structures and tuned for multi-core processors will improve the efficiency of FEA software significantly. An efficient sparse direct solver will increase the speed of linear analysis with multiple right-hand-side (RHS) vectors. In addition, the efficiency of the

time history analysis and non-linear analysis will also improve, where the solution is performed repetitively for different RHS vectors.

1.2 Background

1.2.1 High-Performance Computing

The technology for PCs and Workstations has improved enormously since the development of the first computer. In addition to the consistent increase in the transistor density, major architectural and organizational improvements have occurred. In modern computers, the organization of the computational units allows performing multiple instructions per one clock cycle, and the performance gap between the processor and main memory [4] is hidden with the introduction of high speed memories (caches) between the processor and the main memory. These advancements in computer architecture have dramatically improved performance. Hennessy and Patterson [5] estimated that by 2002, the computers were approximately seven times faster than what would have been without these improvements.

Due to the complex hardware architectures, developing programs which approach the peak speeds of modern processors may be challenging. The computational resources must be used in an optimal fashion to achieve high performance. For example, unnecessary random memory accesses shall be prevented since repeated random memory accesses degrade the performance because of memory access latencies. In addition, programs can be reorganized into blocks of independent instructions to harness instruction level parallelism. Furthermore, SIMD (Single Instruction Multiple Data) instruction sets can be exploited to efficiently perform a single arithmetic operation on multiple data. These optimizations are harnessed in numerical libraries tuned for specific hardware architectures [6-8]. A convenient way to obtain high performance from a computer is to use the libraries tuned for specific computer architectures.

Linear algebra and other numerical operations can be performed efficiently by using the BLAS (Basic Linear Algebra Subprograms) and LAPACK [9] (Linear Algebra PACKage) libraries tuned for a specific architecture. BLAS includes standard subroutines for common vector and matrix operations [10-12]. LAPACK includes subroutines for solution of linear systems, least-square solutions and eigenvalue problems [9]. CPU vendors provide highly tuned BLAS libraries such as MKL [6], ACML [7], and ESSL [8], which can be used to produce high performance programs. In addition to the vendor provided BLAS libraries, other BLAS implementations that are tuned for specific architectures exist. For example, GotoBLAS [13-14] includes optimized BLAS3 kernels that aim to reduce TLB (Translation Look-aside Buffer) misses in matrix multiplication. ATLAS [15] is automatically tuned software that implements BLAS and some of the LAPACK subroutines. In the tuning phase, ATLAS chooses the fastest way to do a BLAS operation among the alternatives [16]. ATLAS also exploits the cache hierarchies of modern processors [16] for a high-performance BLAS.

There are three levels of BLAS subroutines, which are for vector-vector (BLAS1), matrix-vector (BLAS2), and matrix-matrix (BLAS3) operations. Among these, BLAS3 gives the best throughput since the ratio of computations to memory access is largest for BLAS3. A high computation to memory access ratio is beneficial to cache-hierarchies of the modern processors. For sufficiently large matrices, optimized BLAS3 kernels can run at a speed close to peak machine speed [13, 17]. Other benefits of using BLAS libraries are robustness, portability, and readability of the code [18].

We perform numerical experiments to illustrate the high performance of the BLAS3 kernels. Figure 1.1 shows the performance of the MKL BLAS subroutines for an Intel quad-core processor. Only one core is used for the results shown in Figure 1.1 , and the clock speed of a core is 2.4 GHz. SIMD (Single Instruction Multiple Data) instructions allow performing four double precision floating point operations per clock cycle, therefore, each core can potentially perform 9.6 billion double precision floating

point operations per second (9.6 GFlop/sec or 9600 MIPS). This is an upper limit found by assuming that there are no cache misses and no memory latencies. As shown in Figure 1.1, BLAS3 matrix-matrix multiplication runs at a speed close to the peak performance of a single core. On the other hand, the peak speeds for BLAS1 and BLAS2 are about one-half of the machine peak speed. Furthermore, BLAS1 and BLAS2 speeds decrease as the size of the matrix and vector increases. BLAS1 speed starts decreasing as the total memory for vectors exceeds the capacity of the L1 Cache (32 Kbyte). BLAS2 speed, on the other hand, starts decreasing as the memory for the matrix exceeds the capacity of the L2 Cache (4096 Kbyte). As shown in Figure 1.1, BLAS3 gives a sustained performance for a wide range of matrix sizes. Therefore, the performance of BLAS3 is typically more predictable compared to BLAS1 and BLAS2, which is a desirable feature for estimating workloads in order to have a balanced workload assignment.

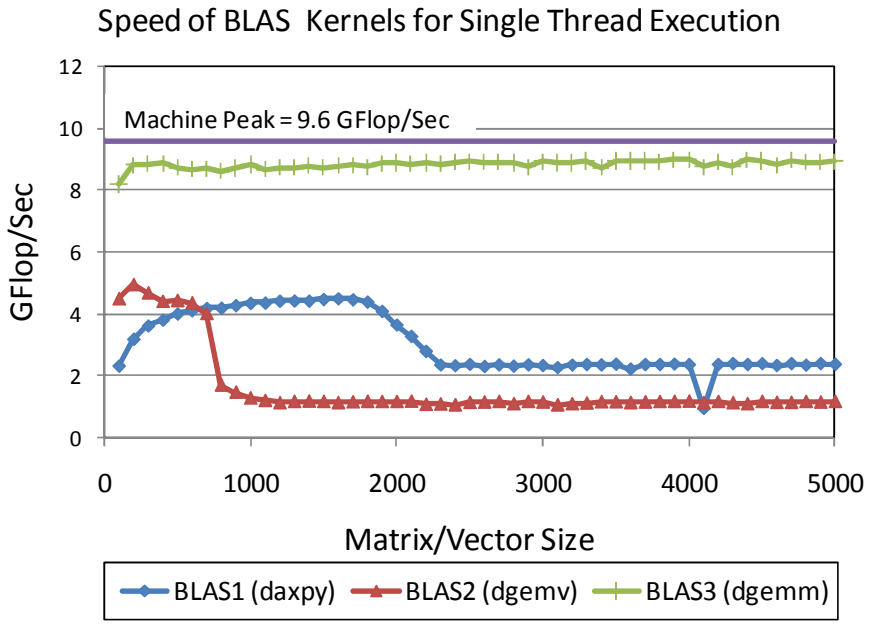


Figure 1.1: The performance of MKL BLAS subroutines. Test runs are performed on Intel Core 2 Quad Q6600 using only one CPU core.

1.2.2 Multi-core and Many-core Processors

Recently, major CPU manufacturers announced that the demand for increased performance will be met by packing multiple processing units into a chip in addition to the improvements in the performance of a single processing unit [1-2]. While automatic performance improvements can be obtained due to increased clock speeds and past architectural advancements, programs must be restructured to harness full computational power of the multi-core processors. This requires redesign of the software developed for uniprocessors which involves finding code segments which may be executed in parallel. Furthermore, the software developers shall consider the architecture of the machine to fully benefit from the computational power introduced by the multi-core processors.

First generation multi-core processors are a subset of symmetric multiprocessing (SMP) architecture. Initially two identical cores are integrated onto a single die. The cores can share some circuitry such as L2 caches and front side bus (FSB). The architecture of a dual-core machine is depicted in Figure 1.2. Here, both processors can access the main memory at high speeds. Furthermore, synchronization and cache coherency can be performed efficiently since the cores are on the same die. All cores have direct access to the main memory via the shared FSB. Therefore, any task can be scheduled efficiently at any processor independent of the location of the data used by the task.

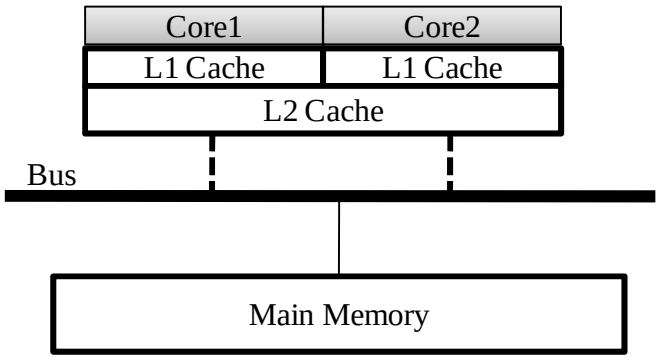


Figure 1.2: Example SMP dual-core processor. Cores access the memory via the shared bus.

The SMP architecture is suitable for multi-core machines with a small number of cores. As the number of cores in the system is increased, the shared bus quickly becomes a bottleneck. Main memory connected to each core with the shared bus can serve only one processor at a time. Consequently, the cores starve of data if they all try to access the memory at the same time. Memory can be organized differently to solve this problem. Non-uniform memory access (NUMA) is an alternative way to organize the memory for multiple processors. In NUMA machines, each processor has local memory that can be accessed in a fast fashion. In addition, non-local memory can be accessed via a special interconnection. However, access to a non-local memory location is slow. An example system with NUMA architecture is shown in Figure 1.3. Here, four cores have a fast connection to the local memory and a processor can access a non-local memory bank via the special interconnection, which is slower than an access to the local memory.

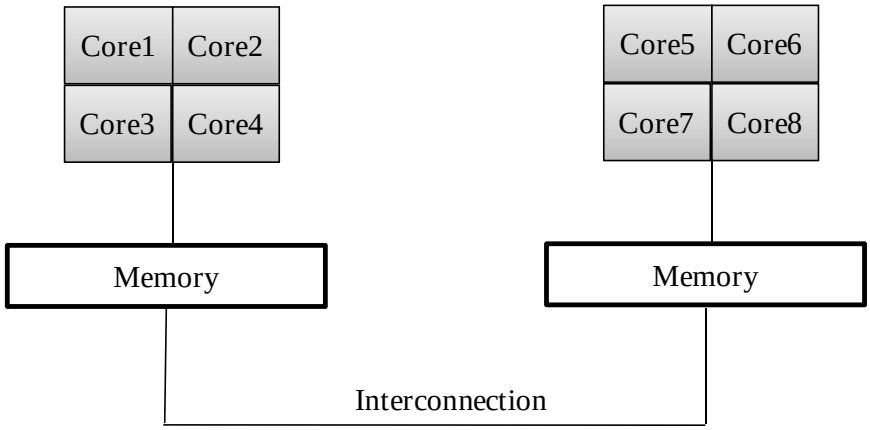


Figure 1.3: A possible architecture for a NUMA system

The exploitation of data locality is crucial to obtain high performance with the NUMA architectures. The tasks shall primarily access the data on the local memory. An access to a non-local memory shall be restricted since the processors remain idle until the data arrives from a non-local memory location. Therefore, programming for NUMA is similar to programming for clusters for which the data used by each task is distributed among processors.

Asanovic et al. [19] discussed the challenges for future many-core systems and gave recommendations about emerging multi-core processors. They stated that programming models shall make it easy to write efficient parallel programs and the focus of a programming model shall be to increase programmer productivity for highly parallel architectures. They also stated that it is important to develop software that can be tuned automatically for multi-core architectures. Automatically tuned software already exists for single core systems. For example, Vuduc et al. [20-21] provided automatically tuned sparse matrix vector multiplication and triangular solution kernels for single core systems. For emerging multi-core architectures, Williams et al. [22] discussed the auto tuning of sparse matrix kernels.

Hill and Marty [23] discussed the importance of speeding up serial code in the multi-core era. Their results were based on the Amdahl's law for symmetric and asymmetric multi-core chips. All processing units are identical in a symmetric multi-core chip, whereas, the computational power/capability of the processing units is not uniform for an asymmetric multi-core chip. They stressed the greater potential of asymmetric multi-core chips compared to symmetric chips for obtaining high performance from architectures with many-cores. They stated that the serial portion of the code will quickly become the bottleneck for many-core chips, and systems with the combination of cores with high serial performance and less powerful parallel cores will scale better since high performance cores can prevent the serial code from becoming the bottleneck. Balakrishnan et al. [24] stated that the asymmetry is beneficial to improve the performance of serial portions of the code. They also stated that current software neglects the asymmetry and software developers typically develop their code for symmetric cores. This negatively impacts the workload balancing for asymmetric processors since some cores are faster than the others. Balakrishnan et al. [24] stated that the applications shall be aware of the asymmetry and shall dynamically adapt to the computing resources. Kumar et al. [25] discussed that the asymmetric cores are more adaptable to different

workloads. Moreover, they showed that asymmetric cores are more energy efficient compared to symmetric cores.

GPGPUs (general purpose computing on graphics processing units) allow asymmetric computing with desktop and laptop computers. GPUs (graphics processing units) are massively parallel architectures with hundreds of cores. Today, peak performance of GPUs is greater than several terra flops for single precision floating point operations. Furthermore, GPUs can also perform double precision floating point operations. GPUs can be programmed easily using frameworks such as CUDA [26-27] and OpenCL [28]. CUDA implements a subset of BLAS kernels, which allows the easy use of GPUs for numerical applications. Garland et al. [29] discussed the effective use of CUDA for speeding up various scientific applications that have data-parallel algorithms.

Figure 1.4 shows an example asymmetric multi-core processor. The system has dual socket quad-core processer and two GPUs. Here, in addition to the computational units being asymmetric, the interconnection of the computational units can be asymmetric also. Asymmetry aware algorithms are crucial to obtain high performance from such systems.

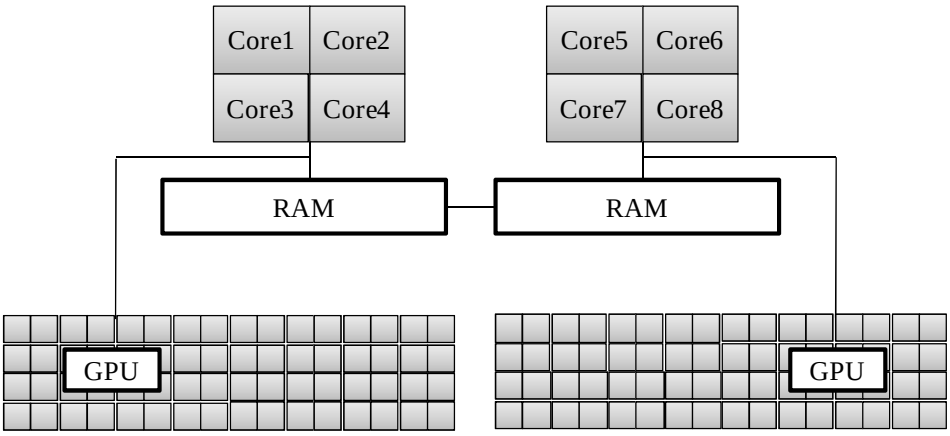


Figure 1.4: Example heterogeneous multi-core processors. The system has two sockets with quad-core processors and two GPUs available for general purpose computing.

Multithreaded BLAS subroutines in MKL[6] can be used to obtain high performance from today's SMP multi-core processors. The LAPACK implementation in MKL relies on multithreaded BLAS [30]. Buttari et al. [30] stated that the use of multithreaded BLAS is not enough for a scalable LAPACK. They proposed parallel linear algebra algorithms that exploit parallelism at a lower level. A dense matrix was divided into square tiles and a directed acyclic graph was used to represent the algorithmic dependency between the tasks associated with the tiles. A critical path was identified in the dependency graph and high priorities were assigned for the tasks on the critical path. The tasks were scheduled asynchronously and dynamically. Tiled algorithms scale better than the ACML [7] counterparts for Cholesky, QR and LU factorizations. Currently, the tiled algorithms are implemented in the PLASMA project [31].

Ltaief et al. [32] discussed a scalable dense Cholesky factorization algorithm that uses both CPUs and GPUs. The transfer rate can be a bottleneck for high performance computations on GPU since the data is first allocated in the main memory by CPU before transferring it to GPU. Then, the data is transferred via a shared connection. Ltaief et al. [32] used static scheduling to minimize the data transfer between the CPU and GPU. This hybrid Cholesky factorization, which uses both CPU and GPU, ran significantly faster than the CPU only counterpart. The source file for their algorithm can be obtained from the MAGMA project website [33].

Finally, Hill and Marty [23] illustrated the high potential of the dynamic multi-core chips that can utilize the cores either in serial mode or in parallel mode. Ipek et al. [34] presented a reconfigurable multiprocessor where independent processors can be used to form a processer with higher serial performance, or they can be used in parallel as needed at runtime.

1.2.3 Solution of Linear System of Equations

The system of linear equations arising from a linear solution of a structure with *n* degree of freedoms is written as:

$$[\mathbf{K}]\{\mathrm{d}\} = \{\mathrm{f}\} \tag{1.1}$$

where **K** is the *n* by *n* stiffness matrix, **f** is a vector of size *n*, which stores the loading at each degree of freedom (dof), and **d** is a vector of size *n*, which stores the unknown displacements corresponding to the loading. If the structure is subjected to multiple loading conditions, *d* and *f* are both *n* by *nrhs* matrices, where *nrhs* is the number of right-hand-side vectors, which is equal to the number of loading conditions.

There are mainly two methods for solving a system of linear equations: iterative and direct methods. Iterative methods are scalable and require less memory compared to direct methods, which make them a better choice for solving very large problems with limited computational resources. However, the convergence of iterative methods depends on the preconditioner used for a problem, and the execution time is unpredictable due to their iterative nature. Additionally, iterative methods can be inefficient for analyzing structures with multiple load cases since the entire solution must be restarted for each RHS vector.

The direct methods, on the other hand, factorize stiffness matrix and once the factorization is complete, the system of equations can be solved efficiently for multiple RHS vectors by forward elimination and back substitution. The sparsity of the system is used to minimize the arithmetic operations and data storage required for the solution. These methods have high numerical precision and guarantee the solution within a predictable amount of time if computational resources are adequate. Because of these advantages, direct methods are preferred in most linear structural analysis software.

1.2.4 Direct Methods for Sparse Linear Systems

The oldest direct method is LU factorization [35], where the matrix **K** is factored into lower and upper triangular matrices, **L** and **U**. If **K** is symmetric and positive definite, the factorization can be written as $\mathbf{LL}^T$, which is called the Cholesky factorization. Cholesky factorization requires fewer arithmetic operations compared to the LU factorization. For indefinite symmetric matrices, the root-free Cholesky factorization is suitable, where the decomposition can be written as $\mathbf{LDL}^T$ [18]. Finally, orthogonal methods such as QR factorization are used for solving least square problems [35].

Only non-zero entries are stored during the sparse factorization and the sparsity should be preserved for an efficient factorization. The nonzero entries in the matrix can be stored using several different schemes. Some common storage schemes are band matrix, profile matrix (also called skyline, variable band, and envelope matrix), element matrix representation, and packed sparse vectors representing the columns and rows of a sparse matrix [36]. Figure 1.5 illustrates variable band and band storage schemes for a simple problem with 2D quadrilateral elements. There is no single storage scheme that performs well for all types of problems. Some storage schemes may be suitable for problems with certain characteristics. For example, the band storage scheme is suitable for long and narrow meshes.

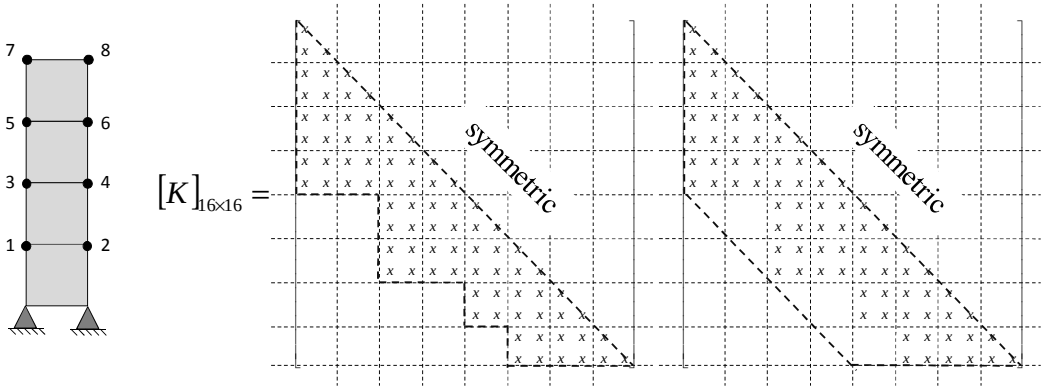


Figure 1.5: Variable band and band storage scheme for an example problem.

The storage scheme determines the algorithms used in different phases of a sparse direct solver. A typical sparse direct solver is composed of four phases [37]:

1. Preprocessing phase – determines pivot ordering which minimizes the time and storage needed by the solution
2. Analysis phase – determines the memory requirements and constructs the data structures that will be used in the factorization and solution phase
3. Numerical factorization phase – determines the factors, **L**
4. Triangular solution phase – performs forward elimination and back substitution using the factors, **L**, found in numerical factorization phase.

If the matrix is positive definite, the phases of a solver are distinct and a pivot ordering found in preprocessing phase can be used for the numerical factorization without numerical stability concerns. For indefinite matrices, on the other hand, a pivot ordering determined in the preprocessing phase may be altered in order to ensure the numerical stability.

A general approach for the solution of sparse matrices is to use sparse data structures throughout the direct solution including the inner loops of the factorization and solution. This reduces the performance since arithmetic operations on sparse vectors require indirect addressing. The drawbacks of indirect addressing are explained by Dongarra et al. [18] using the FORTRAN code shown below:

```
DO 10 I = 1,K
  W(I) = W(I) + ALPHA * A(ICN(I))
10  CONTINUE
```

Here, the entries of the packed sparse vector A are accessed by the indices stored in the vector ICN. The compilers usually have no knowledge about the indices stored in the vector ICN. Therefore, the loop cannot be reorganized for high instruction level parallelism, i.e., the compiler cannot unroll the loop. In addition, if values are assigned to

A(ICN(I)), automatic parallelization will not take place since repeated reference may be made to the same memory location.

The indirect referencing can be reduced if the columns with common sparsity pattern are treated as a single pivot block. Pivot blocks with same sparsity pattern are called supernodes. Indirect referencing is avoided for the factorization operations within a supernode. Figure 1.6 illustrates the supernodes for the example problem shown in Figure 1.5. Here, the 16x16 stiffness matrix is partitioned into 4 supernodes. The supernodes can be partitioned into dense matrix blocks and arithmetic operations on dense matrix blocks can be performed using tuned BLAS3 kernels.

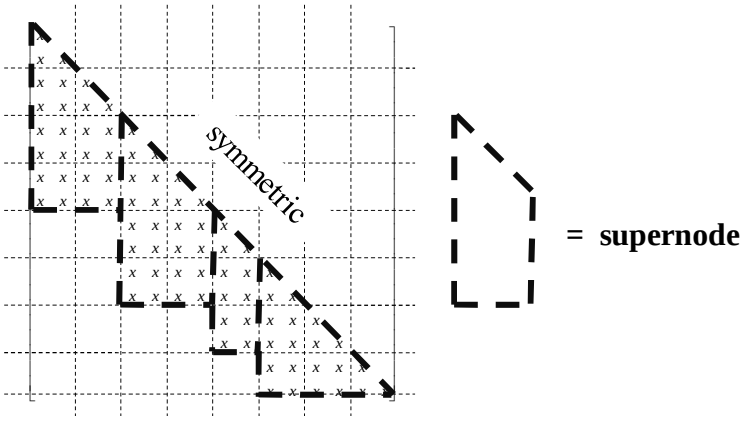


Figure 1.6: For the example problem in Figure 1.5, the supernode partitioning of the stiffness matrix. Supernodes are marked with thick dashed lines.

A direct factorization scheme regardless of whether it uses supernodes or columns as pivots can be classified according to the direction of the column updates. In a left-looking (fan-in) algorithm, a pivot receives updates from the previous pivots just before the factorization of the pivot. In a right-looking (fan-out) algorithm, as soon as the factorization of a pivot is finished, multipliers are calculated and the subsequent pivots are updated immediately. In other words, while a right-looking algorithm accesses to data on the right of a pivot, a left-looking algorithm accesses to data on the left. The left-looking schemes are usually more efficient compared to the right looking schemes since

their memory access patterns are more cache friendly [38-39]. Nevertheless, the performance difference between the factorization schemes is insignificant compared to the benefits obtained by blocking the columns of a coefficient matrix (supernodal schemes) [40].

Band and profile solvers are employed for the band and profile matrix storage schemes respectively. These schemes avoid the indirect referencing, however the efficiency depends on band or profile minimization algorithms. The Cuthill-McKee [41] algorithm is widely used for band minimization. The Reverse Cuthill-McKee [42], Gibbs-Poole-Stockmeyer [43] and Sloan's method [44] can be used for profile minimization. Compared to packed storage schemes, more nonzero entries may be stored in the factors of the band or profile matrices. The main advantage of the band and profile schemes is that the storage scheme is preserved if no column interchanges, i.e., pivoting for numerical stability, are performed during the factorization [36].

For an element matrix representation of a stiffness matrix, a frontal solver [45] may be suitable, in which a subset of elements is assembled to a frontal matrix and factorization is performed for fully assembled dofs. The arithmetic operations are performed on dense matrices and only a small portion of the stiffness matrix is kept in the memory. Similar to band or profile solvers, the element assembly order is important for an efficient frontal solution. Several element ordering algorithms are proposed [46-50] to reduce the CPU time and storage required for the frontal method.

The indirect addressing is avoided in frontal method since all arithmetic operations are performed on a dense frontal matrix. However, if elements cannot be ordered to have a narrow front width, the space and time complexity of the frontal method may be prohibitive. Duff and Reid [51] extended the concept of a single frontal matrix by allowing to work on multiple frontal matrices at a time. This permitted the use of any fill-in minimization algorithm while retaining the efficiency of dense matrix operations on frontal matrices.

The variable elimination in the multifrontal method is similar to condensation of the elements assembled to the frontal matrix. The fully summed dofs are condensed, and the condensed stiffness matrix is stored for future condensation steps. The condensed stiffness matrix is called the update matrix since it will be used to update the pivots in the later stages of the solver. The assembly sequence of elements and update matrices can be represented by a tree structure, which is called the assembly tree [51]. A parent node in the assembly tree can only be processed after all of its children are assembled to the corresponding frontal matrix.

Figure 1.7 shows a sample assembly tree for a condensation sequence of a sample 4x4 mesh. Leaf nodes of the assembly tree represent the finite elements, whereas, the internal nodes represent the intermediate elements (super-elements) obtained by the assembly and condensation of the children nodes. Once all children of an element are processed, its stiffness matrix can be assembled and fully summed dofs of the element can be condensed. The assembly tree shows the dependency between the factorization and triangular solution operations. Factorization at a parent node depends on the factorization of its children. The dependency between the tree nodes is the same for forward elimination. The order is reversed for back substitution where operations at the children nodes depend on the operations at the parent.

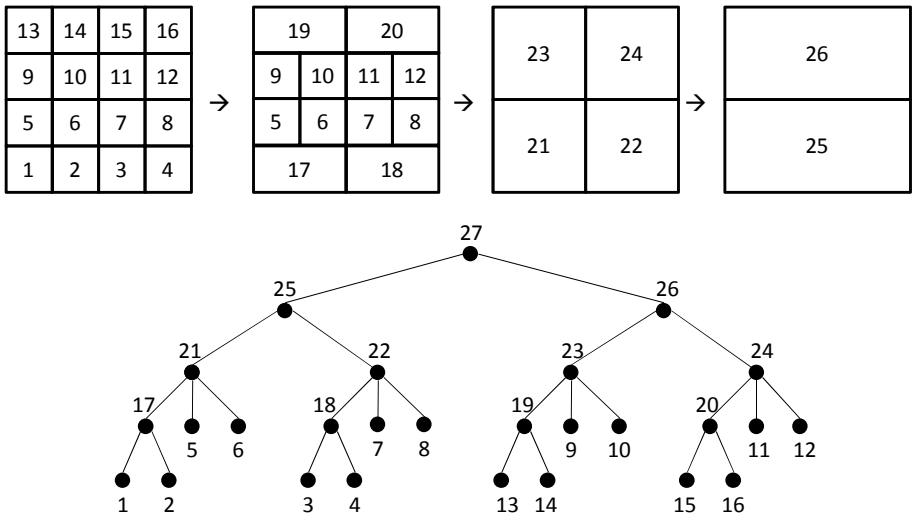


Figure 1.7: The assembly tree for a condensation sequence of a sample mesh

A postorder traversal of an assembly tree allows using a stack data structure to store the update matrices efficiently. The multifrontal method requires a working space for storing the stack of update matrices and the frontal matrix. The maximum value of this working space throughout the course of the factorization is the working storage requirement of the multifrontal method [52]. The working storage can be significant for 3D problems, and it may be larger than the factors for linear programming problems [53]. The traversal of the assembly tree affects the minimum storage required for the multifrontal method. Liu [54] proposed an optimal postordering of the assembly tree that gives the minimum working storage if the frontal matrix of a parent element is created after the elimination operations at its children. Guermouche and L'Excellent [55] extended his work by allowing allocation of the frontal matrix of a parent element after the elimination of any children. Alternatively, the working storage can be reduced by altering the structure of the assembly tree [56].

In the multifrontal method, all algebraic operations are performed on a dense frontal matrix. The cost of using efficient dense matrix operations is the assembly steps performed at the nodes other than the leaves of the assembly tree. These are extra data movement operations since a stiffness matrix can be formed alternatively by assembling

the original finite elements at once. Several algorithms are proposed to reduce these extra assembly operations by merging a parent node with its children according to some amalgamation criteria. Duff and Reid [51] proposed an algorithm that merges a parent node with its eldest child in the postorder traversal if the number of the eliminated variables for both of them is below a certain amalgamation parameter. Ashcraft [57] used an amalgamation criterion that allows merging a parent element with any of its children if the number of logical zero entries introduced by merging is below a certain limiting number. Although the number of floating point operations increases with the node amalgamation, the benefits obtained from the reduction of assembly operations that typically use indirect addressing generally overcomes the increased number of floating point operations.

The space and number of arithmetic operations required for the factorization is minimized by employing a fill-in reduction algorithm in the preprocessing stage of sparse direct methods. These algorithms seek a pivot ordering that minimizes the number of fill-ins introduced to the factors. There are mainly two approaches for reducing fill-ins: local ordering and global ordering. Local techniques are greedy algorithms that choose the best alternative for the next pivot based on some heuristic function. Global techniques, on the other hand, recursively find pivot columns that decouple the rest of the columns in a coefficient matrix, when the pivot columns are removed from the coefficient matrix. The two techniques can be combined to produce hybrid algorithms. Duff and Scott [58] and Gould and Scott [59] reported that the hybrid orderings produce better orderings for large scale test problems used in their numerical experiments.

The number of arithmetic operations required for the factorization of a pivot column is proportional to the square of the number of non-zero entries below the diagonal of symmetric coefficient matrix. To limit the amount of arithmetic operations required for the factorization, the next pivot can be chosen among the columns with minimum number of non-zero entries. This local technique is called the minimum degree ordering

algorithm. Several improvements are made to improve the quality of the orderings and runtime of the original algorithm. George and Liu [60] provided a detailed summary of these improvements. Liu [61] improved the runtime of the algorithm by allowing multiple elimination of the pivots that are from independent sets of low degree columns. Elimination of these pivots will not affect the degree of each other therefore costly degree update operations can be postponed. This algorithm is commonly referred to as multiple minimum degree (MMD) algorithm. MMD gives identical quality orderings in less time compared to the original minimum degree ordering algorithm.

In a minimum degree algorithm, the tie-breaking strategy for the matrix columns with same degree affects the quality of the ordering produced. Usually the minimum degree algorithm is implemented to choose the pivot according to their initial ordering. Lui [60] proposed preordering the nodes with Reverse Cuthill-McKee to improve the quality of the orderings produced by minimum degree algorithm. Furthermore, compared to random permutations, the row-by-row initial ordering of a 2D grid produced higher quality orderings [60]. Cavers [62] evaluated several tie-breaking strategies and recommended using the deficiency information to produce orderings consistently better than an arbitrary selection.

Amestoy et al. [63] proposed an alternative minimum degree heuristic that uses the approximate degree information of the matrix columns instead of the exact degrees. The algorithm is called approximate minimum degree, AMD, ordering and AMD ordering runs significantly faster since computing the approximate degree is computationally cheaper than computing the exact degree. In addition, the quality of the orderings is comparable to the ones produced by MMD.

In addition to the minimum degree heuristics, heuristics that attempt to minimize the fill-ins have been studied. Rothberg and Eisenstat [64] and Ng and Raghavan [65] reported that heuristics based on the minimization of the local fill-ins produce better

orderings compared to minimum degree heuristic. However, the execution time of these algorithms is typically worse than the algorithms based on the minimum degree heuristic.

The global ordering techniques are top-down sparse matrix ordering algorithms that perform domain decomposition to find pivots that do not introduce fill-ins to each other. This scheme for reducing the fill-ins was first proposed by George [66], which is known as nested dissection. George proved that the nested dissection yields operation counts asymptotically close to the optimal orderings for 2D grid problems. However, the runtime of the nested dissection is significantly worse than the minimum degree algorithms, making a complete nested dissection prohibitive for the matrix-orderings. Therefore, instead of the original nested dissection algorithm, an incomplete nested dissection algorithm is used in the hybrid ordering techniques.

In general, the hybrid approach uses graph partitioning techniques such as nested dissection to keep the number of fill-ins low at the higher levels of the assembly tree. After a certain number of nested dissections, a local ordering technique such as minimum degree is used to order the partitions obtained with nested dissections. Finally, the separators found in the nested dissection steps are ordered. Several improvements are made to the hybrid matrix ordering approach, which can be found in references [67-71].

The choice of fill-in minimization algorithm (also called as matrix ordering) influences the efficiency of the numerical factorization and triangular solution phases. The choice is usually between minimum degree algorithms, which are simple and fast, and hybrid orderings, which are more elaborate and usually produce better orderings. Duff and Scott [58] summarized the automatic selection strategy of some direct solver packages. They also proposed their own strategy for selecting the optimum node ordering algorithm. Observing that the minimum degree algorithm gives satisfactory results for small and very sparse matrices, they used experimental criterion based on the size of the coefficient matrix and average number of entries in the columns. For single or small number of factorizations and solves, their strategy produced better total execution time

compared to the strategy of choosing the best ordering algorithm among several alternatives executed for the coefficient matrix. Their work did not consider fill-in minimization heuristics.

Finally, Boman et al. [72] discussed recent advances in parallel partitioning, load balancing and matrix ordering. They employed hypergraphs to represent sparse matrices and demonstrate the superiority of an alternative coarsening scheme for multilevel matrix orderings.

1.2.5 Parallel Direct Solution

According to Dongarra et al. [18], three levels of parallelism are available for the sparse solvers: system level, matrix level, and sub-matrix level. First, system level parallelism is exploited by subdividing the underlying problem into loosely coupled components. This is called substructuring in structural analysis. Second, matrix level parallelism is achieved by exploiting the sparsity pattern of a coefficient matrix. Nested dissection is used to reduce the factorization and solution of the matrix into independent steps. Third, sparse matrix operations can be performed as a series of dense matrix operations on submatrices. Then, the parallelism at the submatrix level can be exploited by using parallel versions of the Level 3 BLAS. These kernels are distributed by most of the CPU vendors, which are optimized for a specific parallel architecture. Most modern sparse codes use BLAS kernels for a scalable and portable implementation.

The parallelism due to the sparsity can be represented by using the elimination tree defined for the sparse matrix [73]. Each node of the elimination tree corresponds to a pivot column in the sparse matrix. The node j in the elimination tree is the parent of node i if j is first entry below the diagonal of the column i of the factors. The computations are independent for the tree nodes that are not ancestors or descendants of each other.

Heath et al. [74] reviewed parallel algorithms for solving sparse symmetric positive definite systems on shared and distributed memory computers. In their paper, the

parallel implementation of all four phases of the solver is discussed. They suggested that although the factorization phase takes most of the time of a solver, the parallelization of the other phases would eventually become important since factorization times are reduced on a regular basis with efficient parallel implementations. Additionally, the solution phase may take considerable time when the system is solved for multiple load vectors.

The structure of the elimination tree can be adjusted to increase the amount of parallelism. Liu [56] proposed a fill-in preserving ordering that uses tree rotations to minimize the parallel completion time. Other fill-in preserving orderings are given in studies [75-76] for optimal parallel factorization times. Nevertheless, Heath et al. [74] suggested that the equivalent tree orderings have minor effect on the amount of parallelism and the node ordering algorithm mainly shapes the features of the parallelism.

A pivot ordering algorithm which merely aims to minimize the fill-in may not be ideal for the parallel solution [74]. Guermouche et al. [77] investigated the effect of node ordering algorithms on the shape of the assembly tree and the memory usage of a parallel multifrontal solver. Conforming to the previous research performed by Gupta et al. [78] and Karypis and Kumar [79], the top-down ordering algorithms provide well-balanced and short assembly trees, which are desirable for parallel processing. Guermouche et al. [77] also reported that deep unbalanced assembly trees produced by the bottom-up heuristics such as minimum degree and minimum fill have fewer memory requirements.

1.2.6 Substructure Level Parallelism

The substructuring is employed to extend the parallelism to the computations before and after the direct solution of the linear systems. Typical computations prior to the direct solution are the calculation of element stiffness matrices, assembling the global stiffness matrix, and constructing the global load vector. Similarly, once the solution is

complete, the results are used in the post-processing stage, in which computations such as finding element internal stresses and element nodal responses are performed.

The parallel implementation of the substructuring method is given in References [80-82]. In the substructuring method, a structural model is subdivided into non-overlapping substructures. The condensation of substructures can be performed independently. Therefore, the parallelism at the condensation phase is trivial. However, there may be a significant data dependency at the interface solution. Some studies [83-84] used a serial solver for the interface problem due to the high data dependency at this level. However, this approach is not scalable since the time required for the interface solution increases with the number of substructures.

For an efficient substructuring, the condensation times of the substructures must be close to each other. This is not an easy problem since the condensation times are unknown before the pivot-orderings for the substructures are found. Hendrickson [85] discussed the lack of expressiveness of graph representations used in the most of the partitioning algorithms. He argued that an undirected graph with weighted edges and nodes is not representative enough for the communication volume between partitions and amount of computation at each partition. Therefore, traditional partitioning strategies such as minimizing the graph edges will not necessarily minimize the communication time between partitions. Similarly, balancing the vertex sums at each partition may not give subdomains with balanced computational loads. Hendrickson and Kolda [86] discussed alternative partitioning approaches and their potential to address the problems with the traditional partitioning approach.

The problems involving adaptive mesh refinement and contact detection have features that change over time. This leads to fluctuations in computational loads of the partitions, which can be tackled by a dynamic load balancing algorithm [87]. Dynamic load balancing adds additional constraint to the partitioning such as minimizing the number of objects moved between the partitions.

Yang and Hsieh [84] proposed an iterative partition optimization method for direct substructuring. After finding the initial partitions, the number of arithmetic operations required for condensation was computed using the symbolic factorization features of the SPOOLES [88]. The weights of the elements were adjusted according to the operation counts found for each substructure. Later, the partitions were modified by using the partitioning packages JOSTLE [89] and METIS [69]. While METIS restarts the partitioning from the scratch, JOSTLE has the feature to adjust the partitions from the previous iterations. Iteratively refining the partitions using JOSTLE generally provided balanced partitions with less iteration. Refining the partitions by moving a small number of elements between the partitions shows similarities to the dynamic partitioning that try to minimize the number of objects moved between partitions.

Kurc and Will [90] proposed a workload balancing scheme for the condensation of the substructures. The METIS partitioning library was used for initial partitioning of the nodal graph representing the structure. Later, the node weights of partitions were adjusted according to the estimated operation counts. The PARMETIS [91] library was used for repartitioning according to the adjusted node weights. The diffusion and scratch-remap repartitioning algorithms were investigated. They found that scratch-remap gives more computationally balanced substructures. Moreover, the number of interface equations was smaller compared to the diffusion algorithm. They also stated that time spent in the repartitioning iterations was insignificant compared to the improvements obtained in the condensation times.

1.2.7 Survey of Sparse Direct Solvers

In past years, a number of direct sparse solver packages have been developed. The capabilities and algorithms used in the solver packages vary. Table 1.1 shows the main features of some of the current public domain direct solvers.

The solver packages usually involve several ordering algorithms available to the user. The user may choose from the ordering algorithms offered by the solver or a custom pivot ordering may be provided. Some packages offer a default strategy for ordering, in case the user is reluctant to make a choice among ordering algorithms. Some packages try several ordering algorithms and choose the one that produce the best results since there is no single ordering algorithm that consistently provides the best orderings in a reasonable time. A better strategy may be to try a computationally cheap local ordering algorithm first, and then running a global ordering algorithm if the local ordering is not satisfactory. CHOLMOD employs a similar strategy by initially ordering the pivots with AMD algorithm. If the quality of the ordering is below a certain criteria or the number of equations is large, a global ordering algorithm is executed.

Code	Algorithm	Matrix type	Platform
CHOLMOD [92-93]	Left	Hermitian	Serial
PASTIX [94-95]	Left	Hermitian	Distributed memory
PSPASES [78, 96]	Multifrontal	Hermitian	Distributed memory
PARDISO [97-98]	Left-right	Hermitian, Symmetric pattern, Unsymmetric	Shared memory
WSMP [99-100]	Multifrontal	Hermitian, Unsymmetric	Shared/Distributed memory
Oblio [101]	Left, Right, Multifrontal	Symmetric	Serial
UMFPACK [102-103]	Multifrontal	Unsymmetric	Serial
SPOOLES [88, 104]	Left looking	Symmetric, Symmetric pattern	Shared/Distributed memory
SuperLU [105-106]	Left	Unsymmetric	Serial
MUMPS [107-108]	Multifrontal	Symmetric, Symmetric pattern	Distributed memory
TAUCS [109-110]	Left, Multifrontal	Symmetric, Unsymmetric	Distributed memory

Table 1.1: Features of the direct sparse solver packages

Gould et al. [111] evaluated the performance of state-of-the-art direct solvers for symmetric matrices. The solvers were executed on a single processor for matrices of order greater than 10,000. In their experiments with positive definite matrices, the factorization time of WSMP was usually found to be better than the other solvers. However, the preprocessing phase required significant time for this package since WSMP executes both nested dissection and minimum fill-in algorithms and the pivot-ordering with the lower number of non-zeros is chosen. In general, the factorization times are closely related to the number of non-zero entries in the factors, and the difference between the factorization algorithms (left-looking, right-looking or multifrontal) was not very significant. The total solution time was also compared for a single right hand side vector. Balancing the time spent in all phases of the solver, CHOLMOD produced the best overall performance. The time spent in the preprocessing phase of WSMP reduced its overall performance. PARDISO gave the second best factorization and overall solution time among direct solvers evaluated by Gould et al. [111].

The serial and parallel performance of some solvers were experimentally evaluated on 24-processor IBM RS6000 by Gupta and Muliadi [112]. The relatively new solvers, MUMPS and WSMP, performed significantly better, usually by an order of magnitude. The performance difference was related to the efficient use of the Level 3 BLAS kernels in the more recent solver packages.

The performance of two distributed memory solvers, MUMPS and SuperLU, was evaluated by Amestoy et al. [113]. Allowing artificial non-zero entries to increase the size of the dense matrix blocks, MUMPS performed better than SuperLU. Efficiency of the BLAS3 kernels usually offsets the additional computations as a result of the extra zero entries stored in the larger dense matrix blocks used by MUMPS. SuperLU uses complex data structures to store the nonzero entries only, which may be beneficial for matrices with irregular sparsity pattern. MUMPS performed better for small number of processors since it has lower communication overhead. However, SuperLU was expected to be more

scalable since it exploits the parallelism better with the cost of larger communication demands.

There is no recent study that compares the performance of the state-of-the-art solvers on modern multi-core processors. Moreover, previous studies evaluated the solver performance for the sparse matrices from different fields, not particularly focusing on the problems arising from structural mechanics. Most of the test matrices used in these studies can be found at University of Florida sparse matrix collection [114] in an assembled format.

In order to use an available direct solver package, the complete stiffness matrix usually needs to be assembled. Among solvers discussed in this section, only MUMPS allows the element matrix representation while all other solvers require a fully assembled sparse matrix. The complete assembly of the stiffness matrix can be prohibitive for large scale problems. Additionally, for some cases, the time for assembling a stiffness matrix can be comparable to the time for the factorization of the stiffness matrix. While evaluating the performance of a direct solver, in addition to time spent in the phases of the solver, the time spent and memory requirements for building the stiffness matrix of a structure must be considered.

1.2.8 Design of the Solver Packages

The primary concern in the design of a direct solver is the performance of the solver. Scott and Hu [37] discussed the features of an ideal solver package, other than the efficiency in terms of CPU time and memory. In addition to a comprehensive documentation aimed for both experienced and inexperienced users, the features of an ideal solver are summarized as:

Simplicity: The algorithmic details of the solver package should be hidden from the user. The interface of the solver package should be designed for a user with no or little knowledge about the sparse linear solver algorithms. Object-oriented programming

techniques can be used to hide the implementation details. Instead of using arrays holding the implementation details, an object can be passed as the single input argument to a phase of the solver. However, the properties of the objects must be accessible for advanced users.

Clarity: The symbolic factorization and numerical factorization phases shall be separated to allow the reuse of a symbolic factorization. Similarly, the solve phase and factorization phase shall be distinct to allow solution for multiple right hand sides.

Smartness: Good default values for the input parameters such as the ordering algorithm, blocking and pivoting strategy shall be chosen automatically. An optional error checking for the input data must be offered.

Flexibility: The experienced user shall have the flexibility to experiment with the different ordering strategies. The information about the matrix inertia, level of accuracy, and nonzero entries in factors shall be accessible.

Persistence: This is ability of the solver to recover from failure. For example, if there is not enough memory, then the user shall be informed with a proper error code. If the solver has an out-of-core factorization option, it can automatically switch to the out-of-core factorization algorithm in case the problem cannot be solved using the main memory only.

Robustness: Iterative refinement shall be performed automatically. The residuals and condition number estimates shall also be provided.

Safety: The solver package should be thread-safe. Additionally, memory leaks should be avoided.

Scott and Hu [37] summarized the interface, documentation, and matrix input format of the state-of-the-art solvers. They concluded that none of the current solvers meets the criteria of an ideal solver.

George and Liu [115] proposed an object-oriented design for the user interface design of sparse matrix package. Objects were used to hide the implementation details of

the functions and a standard interface was provided by overloading the function names for different types of objects. Both standard and experienced users were considered in their design. Two main classes were offered for the standard user: Solver and Problem. Problem class holds the linear system of equations that will be solved. It allows entering, modifying and querying the entries of the sparse matrix and right hand side vectors. Solver class accepts a problem instance as input and produces a solution. The function Solve is overloaded for different types of solver, which calls the functions for the pivot ordering, symbolic factorization, matrix input, factorization, and forward/backward substitution. These functions are accessible for the research purpose and they can be ignored by a standard user. Considering researchers, the design also allows accessing low level objects used in the different phases of the solver such as graphs, pivot ordering, and elimination tree.

For the local ordering algorithms, Kumfert and Pothen [116] provided an object oriented framework. For a minimum priority ordering, they indentified three main classes: quotient graph, bucket sorter, and priority strategy. Quotient graph holds the compressed quotient graph representing the non-zero entries in the coefficient matrix during the factorization, bucket sorter holds item-key pairs sorted according to the key values, and priority strategy represents the local heuristic to choose the next pivot. Their design allows using different priority strategy at different stages of the ordering algorithm. An algorithm that allows multiple eliminations can be used initially, when there are many independent sets of vertices satisfying the pivot selection criterion. As the graph gets tightly connected, approximate degree orderings that typically does not allow multiple eliminations can be employed.

The main struggle in their design was to provide a common interface for the adjacent vertices in the elimination graph. An adjacent vertex in the elimination graph corresponds to a vertex reachable by a path of eliminated vertices in the quotient graph. Those paths are referred to as reachable paths and a set of vertices that are reachable via

such paths are referred to as reachable set of a vertex. Initially, the class ReachableSetIterator provided a standard way to access the reachable sets of a vertex. However, the overhead of the iterator forced them to provide lower level accesses to the adjacency lists of the quotient graph. With the cost of increasing the coupling between the classes, this provided the desired performance. Compared to Lui's minimum degree code, GENMMD [61], their implementation takes more time by a factor of three to four on average. Their design allows measuring the different components of the algorithm and most expensive part of the code was found to be the update for the quotient graph.

Dobrian et al. [101] discussed the design of an efficient object oriented software for direct solution of sparse systems. The solver was developed mainly in C++, however the core factorization operations were optimized with a C or FORTRAN compiler, which were only a small portion of the code but took most of the execution time. For the remaining part of the code, the complexity was tackled using the object oriented design techniques. The main classes are organized into a single inheritance tree with two main branches: DataStructure and Algorithm, which are both derived from the Object class. Each object can print itself and it returns the error status in case of request. The DataStructure provides interface for data validation and resetting the data structure. The Algorithm provides common interface for the execution of every algorithm and it stores the running time of the algorithm. Some algorithm classes, such as the factorization algorithms, are composed of several algorithm objects. The multifrontal factorization algorithm, for example, is composed of algorithm classes for computing the elimination trees, performing symbolic factorization, and performing the numerical factorization.

One of the design requirements of Dobrian et al. [101] is easy integration of the different matrix formats. For this purpose, all matrix classes are derived from a base matrix class that has the sparse matrix data. Each derived matrix class has a constructor that accepts base matrix class and constructs the sparse matrix in the format the class represents. This prevents cyclic coupling between the classes representing different

matrix formats. A similar approach is followed for handling different graph formats and matrices and graphs can be built from each other by using the constructors that accept the base graph and matrix classes.

Ashcraft and Grimes [88] developed an object oriented sparse matrix library, SPOOLES, which supports serial, multithreaded, and MPI environments. For the preprocessing phase, minimum degree, nested dissection and multisection orderings are provided. The library is within the public domain and there are no licensing restrictions.

Sala et al. [117] proposed an interface library between direct solver libraries and the applications that require a direct solver for serial and distributed systems. The interface involves three phases, preordering and symbolic factorization, numerical factorization, and solution. The abstract classes that define the interface are provided. These are a map class representing local to global mapping, a vector class, a square matrix class, a linear system of equations class, and a solver class. A concrete implementation of the interfaces can be found in the webpage of the AMESOS project [118]. The interfaces are provided for the direct solver libraries LAPACK [9], UMFPACK [103], TAUCS [110], PARDISO [98], SuperLU [106], DSCPACK [119], SCALAPACK [120] ,and MUMPS [108].

1.3 Objective and Scope

The main objective of this research is to develop a direct solution procedure which exploits the parallelism that exists in current SMP multi-core processors and is efficient for solving the linear system of equations that occur in finite element problems. In order to accomplish this objective, the following tasks were performed and are further described in subsequent chapters of this study:

- Determine the factors which contribute to the performance of the direct solver

A direct solver is typically composed of four phases: preprocessing, analysis, numerical factorization, and triangular solution. The performance of all four phases

contributes to the overall performance of the solver. Therefore, the performance of each phase is investigated and approaches to improve the performance of these phases are described. The contribution from each phase to the overall execution time is studied. Furthermore, the factors that affect the parallel performance of the solver on SMP multi-core architectures are studied.

- Investigate the performance of matrix ordering algorithms for finite element problems.

As explained in Section 1.2.4, there are various matrix ordering algorithms available in the literature. The matrix ordering algorithm greatly determines the memory and CPU time required for the factorization. The matrix ordering algorithms that yield minimum storage and factorization time requirements are determined. A suite of finite element models are used to determine the optimal algorithms.

- Improve solver performance by incorporating FE model information

A general purpose sparse solver is oblivious to the FE model underlying a linear system of equations. Therefore, the information about the finite element model is not used if a general purpose sparse solver is used for the solution. This information can be useful to improve the performance of a direct solver. For example, the coordinate information of the nodes in the finite element model can be used for an initial node numbering for the matrix-ordering programs. Local matrix-ordering algorithms usually produce pivot-orderings resulting in fewer fill-ins in the factors if this initial node numbering is used. In addition, the general characteristics of finite element models such as multiple dofs at a node and limited connectivity of elements can be exploited to improve the performance of a direct solver. For example, the size of data structures used in the solver package can be reduced significantly by the use of a coarser model formed by merging adjacent elements with each other and by the use of a supervariable graph. This will reduce the space and time required for the matrix ordering algorithms. The

solver package exploits these opportunities to improve the overall performance of the direct solution.

- Implement a high performance solver package which allows experimenting with alternative matrix-ordering and factorization algorithms.

The developed solver package has two primary goals: high-performance of the core factorization subroutines and flexibility in the preprocessing phase. The high-performance of the core factorization subroutines is achieved by employing the BLAS/LAPACK kernels tuned for specific computer architecture. By using the tuned BLAS/LAPACK kernels, a parallel multifrontal scheme is developed for an efficient numerical solution on SMP multi-core processors. The solver package is also maintainable and extensible for easy implementation of emerging preprocessing algorithms for the direct solution of sparse linear systems. Object-oriented design principles are used to allow easy implementation of other preprocessing algorithms.

- Improve the performance of triangular solution phase for multiple RHS vectors.

The design of structures typically includes analysis of FE models for multiple RHS vectors. Therefore, it is crucial to emphasize the efficiency of the triangular solution for multiple RHS vectors. A triangular solution scheme that is especially efficient for the triangular solution of multiple RHS vectors is developed.

- Improve the performance of assembly operations for the stiffness matrix.

As the execution time of numerical solution decreases by the use of parallel algorithms, other components of the FE analysis software will become a bottleneck, such as the assembly of the stiffness matrix. The developed code works with the element connectivity information and element stiffness matrices instead of an assembled global stiffness matrix. In fact, the global stiffness matrix is never assembled, and the assembly operations are interleaved with the factorization steps. In this scheme, the parallelism is

easily extended to the assembly operations since they are interleaved with the numerical factorization operations, which are performed in parallel.

- Develop a performance model for the direct solution of structures on SMP multi-core processors.

It is important to understand the performance of an application for workload balancing on multi-core processors. A performance model is built to predict the performance of the factorization phase on SMP multi-core processors. The execution time of factorization can be predicted by the use of the developed performance model. The estimated performance is compared with the actual performance of the solver. This helps to determine unpredictable performance degradations due to the limited computational resources. Based on the estimated execution times, the parallelism can be exploited in an optimal fashion to minimize the factorization times on SMP multi-core processors. In addition, the performance model is used to choose the pivot-ordering that will minimize the estimated factorization time among alternative pivot-orderings found by different preprocessing strategies.

- Tune the solver package according to the results obtained from the numerical experiments.

Numerical experiments are performed for the solution of FE models. Based on the results from the numerical experiments, the optimal parameters for the algorithms are determined for the solution of FE models. The execution time of the developed code is compared with a state-of-the-art direct solver.

- Develop recommendations for obtaining high-performance from multi-core architectures other than SMP.

Preliminary analyses are performed for heterogeneous multi-core architectures having GPGPUs. Recommendations are developed to obtain high performance from these multi-core architectures.

1.4 Thesis Outline

The remainder of the thesis is organized as follows. Chapter 2 discusses factors contributing to the performance of a direct solver and implementation of the direct solver developed in this study. Performance evaluation methodology is also explained in Chapter 2. Chapters 3-5 discuss the implementation of the preprocessing phase, analysis phase and factorization & triangular solution phases respectively. The algorithms and data structures used in each phase are discussed. Chapters 3-5 also discuss approaches to improve the performance of a direct solver for the solution FE problems. Chapter 6 evaluates the performance of various algorithms for the sparse direct solution of FE problems. Among various alternatives, the algorithms that yield favorable factorization times are presented in Chapter 6. Chapter 7 demonstrates the performance of developed code on a number of test problems. Chapter 8 gives the results from the preliminary experiments for performing factorizations using GPGPUs. Finally, Chapter 9 summarizes the achievements and gives recommendations for future work.

CHAPTER 2

PERFORMANCE, IMPLEMENTATION, AND TESTING METHODOLOGY

In this chapter, the factors contributing to the performance of sparse solvers are discussed. The methodology for improving the performance of the sparse direct solver is explained. Implementation of the solver and test problems are briefly discussed.

2.1 High-Performance Sparse Direct Solution

As discussed in Chapter 1, high performance relative to the machine peak speed can be achieved if the ratio of computations to memory access is high. The ratio of floating point operations required for the factorization (flop) to the number of non-zeros in the factorized stiffness matrix (non-zero) is a measure of computations per memory access. This ratio (flop/non-zero) varies depending on the size of the FE models. Figure 2.1 shows the flop/non-zero ratios for example 2D grids with 4-node quadrilateral elements. As shown in Figure 2.1, the flop/non-zero ratio increases as the model size increases. Therefore, there is the potential to have factorization speeds close to the machine peak speed for sufficiently large FE models. For these models, high performance factorization can be achieved by employing BLAS3 kernels [17]. Chen et al. [92] stated that for sparse matrices with the flop/non-zero ratios larger than 40, the use of BLAS based factorization is more advantageous compared to non-BLAS based factorization. The critical flop/non-zero value is determined based on the numerical experiments shown in Figure 2.2. As shown in Figure 2.2, the BLAS based factorization is significantly faster than the non-BLAS based factorization for problems with large flop/non-zero ratios. For the example FE problems shown in Figure 2.1, problems larger than 20×20 grid have flop/non-zero ratio larger than 40, the critical value given by Chen et al. [92]. The

flop/non-zero ratio is even larger for plane frame problems and 3D models. Consequently, a BLAS based factorization algorithm is suitable for most of the practical FE problems solved with today's computers.

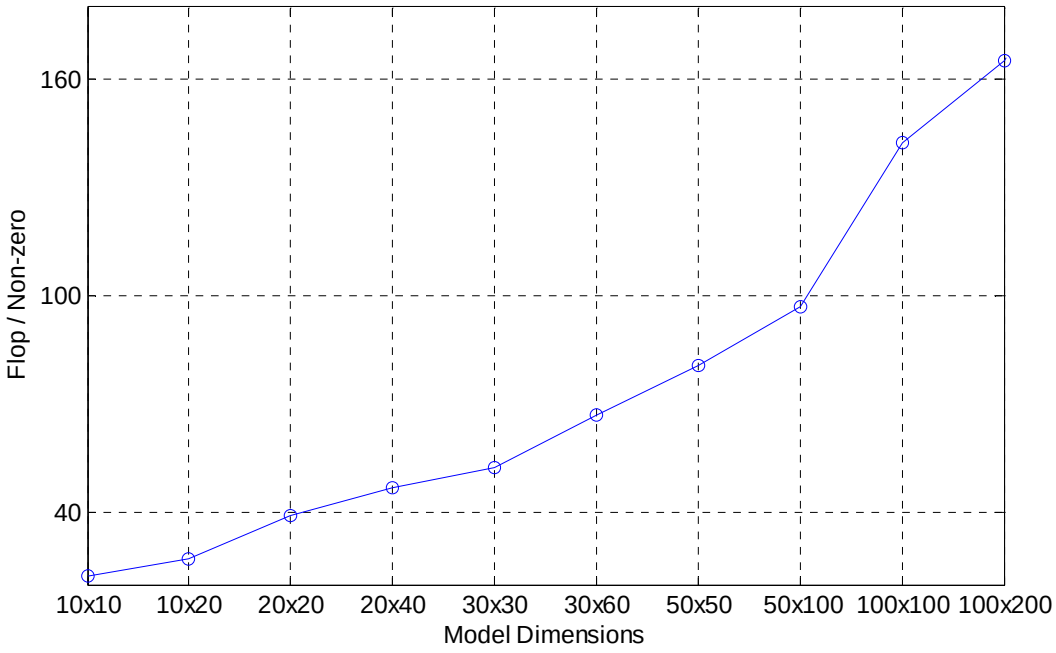


Figure 2.1: The flop/non-zero ratios for 2D grid models with 4-node quadrilateral elements. There are 2 dofs per node.

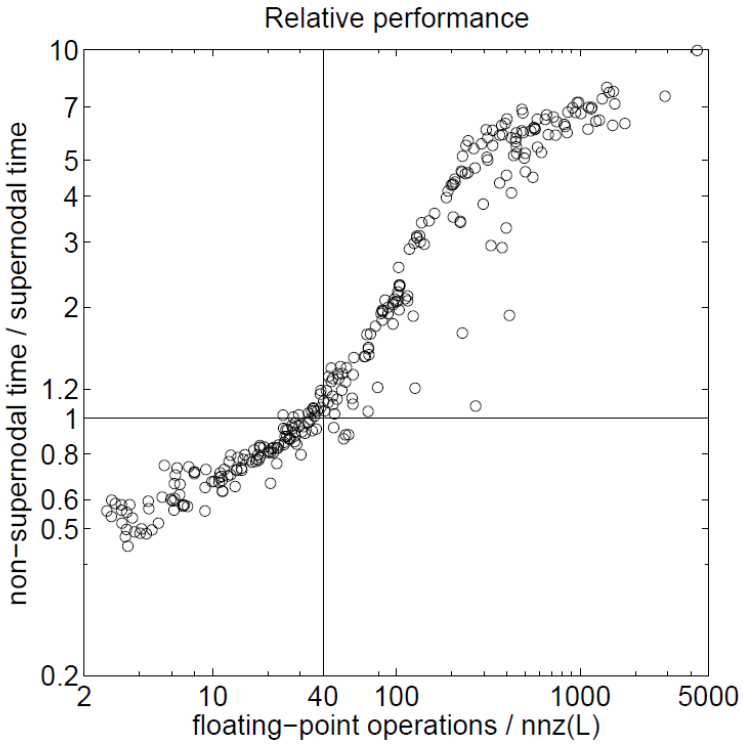


Figure 2.2: For University of Florida Sparse Matrix Collection [114], relative performance of the BLAS based factorization and the non-BLAS based factorization. Figure is taken from Chen et al. [92].

Two main factors contribute to the high performance of sparse factorization: minimization of number of floating point operations required for factorization and maximization of the factorization speed. For the former, a pivot-ordering is found, which aims to minimize the flop and non-zero. For the latter, BLAS3 kernels are employed in order to achieve a high factorization speed. Although factorization operations typically correspond to the largest portion of the overall operation count of a sparse solver, the execution time of the remaining code can be significant due to relatively low speed of handling sparse data structures. An efficient numerical factorization code minimizes the time spend in the subroutines other than the BLAS3 kernels which are employed for the factorization operations. In this study, the factorization speed is further increased by effectively exploiting the parallelism in multi-core computers.

2.2 Other Considerations for Achieving High-Performance

As previously discussed, the solver package consists of preprocessing, analysis, numerical factorization, and triangular solution phases. The main design goal of a high-performance solver package is to minimize the time spent in the solver package. The numerical factorization is typically the most computationally intensive component of a solver package. However, the time spent in the preprocessing and analysis phases may be comparable to the time for numerical factorization, especially for some 2D problems. Moreover, the triangular solution may take a significant amount of time if there are a large number of RHS vectors. Therefore, the time spent in all phases of the solver package must be considered in order to achieve high-performance.

The contributions of the various components of the solver package to the overall execution time are demonstrated using two simple example problems. The overall performance for two matrix ordering programs is illustrated for an example 2D problem as shown in Table 2.1. The time spent at different phases of the solver for the numerical factorization and triangular solution with four threads is given in Table 2.1. The second row of the Table 2.1 shows the solver execution times for a local ordering. For the local ordering, the approximate minimum fill-in (AMF) algorithm in the SCOTCH library [121] is used. The third row of the Table 2.1 shows the solver execution times for the hybrid ordering algorithm in METIS library [69] (HMETIS). The local ordering AMF runs faster than the hybrid ordering HMETIS, but it may produce pivot-orderings that are not suitable for parallel processing. As shown in Table 2.1, the four-thread factorization time for HMETIS ordering is smaller than factorization time for AMF ordering. However, factorization plus matrix-ordering time for the AMF ordering is smaller compared to the HMETIS ordering due to the small matrix-ordering time for AMF. Therefore, the use of AMF ordering will minimize the overall execution time if the factorization and triangular solution are performed only once. However, if the factorization is repeated multiple times, then the factorization time will dominate the

overall execution time. In this case, it may be desirable to use a more time consuming preprocessing algorithm if it helps to reduce the factorization times. For example, the overall time with HMETIS ordering will be better than the one for AMF ordering if the factorization is repeated for more than 8 times for the example problem shown in Table 2.1.

As it is illustrated in the previous example, even though HMETIS execution times are greater than the factorization times, the extra time spent in the matrix ordering phase is worthwhile if the factorization is repeated. Therefore, the type of analysis shall be considered for a strategy that minimizes overall solver time. Namely, if the factorization and triangular solution are performed multiple times, for example, nonlinear analysis, the time spent in the numerical factorization and triangular solution phases may dominate the overall execution time. In this case, additional time can be spent in the preprocessing and analysis phases in order to minimize the numerical factorization and triangular solution times. On the other hand, if the solution is performed only once, the extra time spent in the preprocessing and analysis phases may increase the overall execution time significantly. This is especially true for parallel factorization and triangular solution, for which the contributions of the serial preprocessing and analysis times increase due to the time reductions in the numerical factorization and triangular solution phases.

Ordering Algorithm	**Ordering Time**	**Factorization Time**	**Overall Time (Factorization & Ordering)**	**Solution Time (100 RHS)**	**Overall Time for Solution**
AMF	1.00	3.52	4.52	7.21	11.73
HMETIS	8.09	2.54	10.63	4.40	15.03

Table 2.1: Four-thread execution time of the solver package for alternative matrix ordering programs. Test problem is 50×10000 grid with 2D 4-node quadrilateral elements. All units are seconds.

As shown in the previous example, the execution time of the matrix-ordering program may affect the overall performance for 2D problems with numerical factorization times comparable to the hybrid matrix ordering times. On the other hand, for large 3D problems, the time spent in the matrix ordering program is typically small compared to the time spent in the factorization phase. Therefore, the overall performance is usually governed by the time spent in the factorization phase, and more elaborate matrix ordering strategies can be employed to minimize the factorization time at the expense of extra time spent in the preprocessing phase. Table 2.2 shows the overall performance for an example 3D problem. As shown in Table 2.2, the factorization time usually dominates the overall execution time. For this problem, a matrix ordering strategy that executes alternative matrix ordering programs and picks the best result among the pivot-orderings is more likely to minimize the overall execution time compared to the use of a single matrix-ordering algorithm.

Ordering Algorithm	**Ordering Time**	**Factorization Time**	**Overall Time (Factorization & Ordering)**	**Solution Time (100 RHS)**	**Overall Time for Solution**
AMF	0.05	15.73	15.78	2.60	18.38
HMETIS	0.32	6.41	6.73	1.68	8.41

Table 2.2: Four-thread execution time of the solver package for alternative matrix ordering programs. Test problem is 30×30×30 grid with 3D 8-node solid elements. All units are seconds.

In this study, strategies to improve the overall performance of the solver are discussed. In addition to the numerical factorization and solution phases, the performance of the preprocessing and analyze phases is evaluated. Although parallel implementations of hybrid matrix ordering libraries exist, such as PARMETIS [91] and PT-Scotch [122], all matrix ordering programs are executed in the serial mode. Therefore, the parallel

execution times of the hybrid matrix ordering programs will be less than the serial execution times given in this study.

2.3 Performance Evaluation

The execution time of the numerical factorization depends on two factors: floating point operations (flop) required for factorization and the speed (GFlop/sec) of numerical factorization. Matrix ordering programs aim to minimize the flop and efficient algorithms are employed to perform the numerical factorization and solution as fast as possible. As stated in the previous section, the time spent in the matrix ordering stage can also contribute to the overall execution time of the solver and must be considered when we evaluate the performance of the solver package. The performance of the solver package is evaluated using the following performance criteria:

- Non-zero – number of non-zero entries in the lower-diagonal after the factorization.
- Factorization flop – number of floating point operations required for Cholesky decomposition.
- Solution flop – number of floating point operations required for numerical solution.
- Ordering time – execution time of the matrix ordering strategy.
- Factorization time – execution time for the numerical factorization.
- Solution time – execution time for the numerical solution.
- Overall time – total time required for the matrix ordering strategy, numerical factorization and numerical solution.

The first three performance criteria are theoretical values and are calculated at the preprocessing and analysis phases prior to the factorization. The remaining performance criteria are found by measuring the execution time of different components of the direct solver. All execution times are wall clock times. The matrix ordering programs typically

aim to minimize the non-zeros introduced to the factors. The non-zero performance criterion can be used to compare the results from different matrix ordering programs. Furthermore, the non-zero is a measure of memory required for the factorization. Factorization flop is the theoretical operation count required for the numerical factorization. Similarly, solution flop is the theoretical operation count required for the numerical solution.

For all performance criteria, a smaller value is better. A value for a performance criterion is compared with the smallest performance criterion obtained for a test problem. Two types of plots are used to compare the performance of alternative configurations: normalized performance plots and performance profiles [123].

Normalized performance plots present the performance of configurations normalized according to the best results for all test problems used in the performance evaluation. Normalized plots help to see how a configuration performs for a test problem compared to the configuration that gives the best results. For the normalized performance plots, if the value on the y axis is 1.0 for a configuration, then that configuration gives the best results for the problem given on the x axis. When there are a large number of test problems, we may want to compare the overall performance of the configurations rather than the performance for individual test problems. The performance profile proposed by Dolan and More [123] is used to evaluate the overall performance of alternative solution strategies for a test suite. For a given performance criterion, $p_i(\alpha)$ is the performance profile of the i^{th} configuration, which is typically a solution strategy. For i^{th} configuration, $p_i(\alpha)$ gives the percentage of the test problems that is within the α times the best performance criterion. For example, $p_i(1)$ gives the percentage of test problems for which the i^{th} configuration produces the best results. Similarly, $p_i(2)$ gives the percentage of test problems that the i^{th} configuration produces results within two times the best result. By definition, $p_i(\alpha)$ is a non-decreasing function. The performance profile has been used to

evaluate the performance of direct solvers and matrix ordering strategies in previous studies [58, 111].

The performance plots are demonstrated using an illustrative example. Hypothetical performances of three different configurations are given in Table 2.3. Here, Configuration 3 gives the best performance for all models except from Model 1. Figure 2.3 shows the normalized performance plots. Here, the superiority of Configuration 3 for three out of four models is apparent. Figure 2.3 also shows that the Configuration 2 gives the best result for the model that Configuration 3 fails to give the best result. The relative performance of the configurations can be compared for each model by using the Figure 2.3. The performance profiles for the example configurations are shown in Figure 2.4. Figure 2.4 shows that Configuration 3 gives the best results for 75% of the models and Configuration 2 gives the best performance for the remaining 25%, which is only one problem in our example. Figure 2.4 also shows that the performance of Configuration 1 and Configuration 2 is similar for 75% of the models. For the remaining 25% of the models, Configuration 1 performs better than Configuration 2.

Configuration No.	**Model 1**	**Model 2**	**Model 3**	**Model 4**
1	2.0	1.5	2.5	3.0
2	**1.75**	1.75	2.25	3.5
3	2.5	**1.40**	**2.0**	**2.0**

Table 2.3: An illustrative example for performance plots. The values for a performance criterion are shown for three configurations and four models.

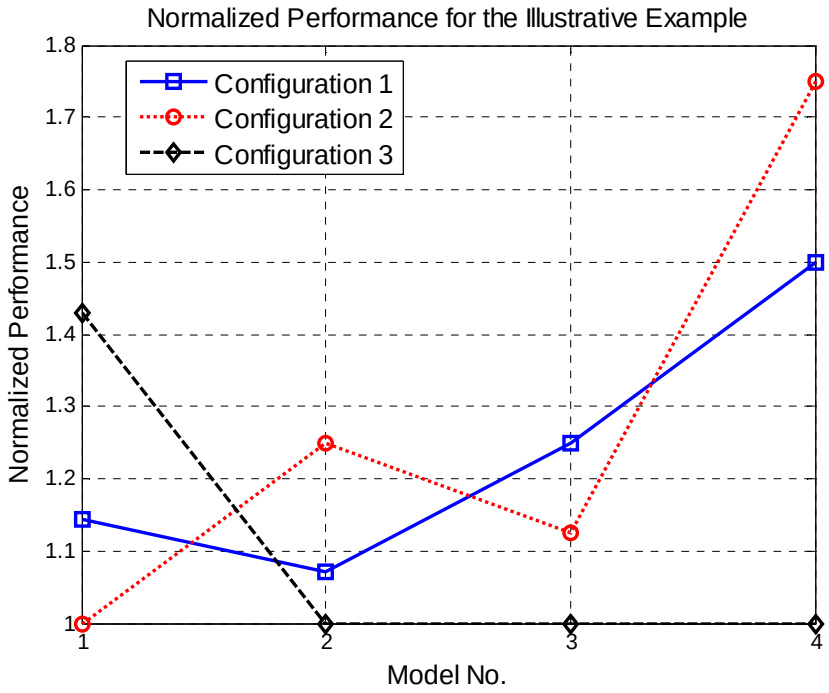


Figure 2.3: Normalized performance plots for the illustrative example given in Table 2.3.

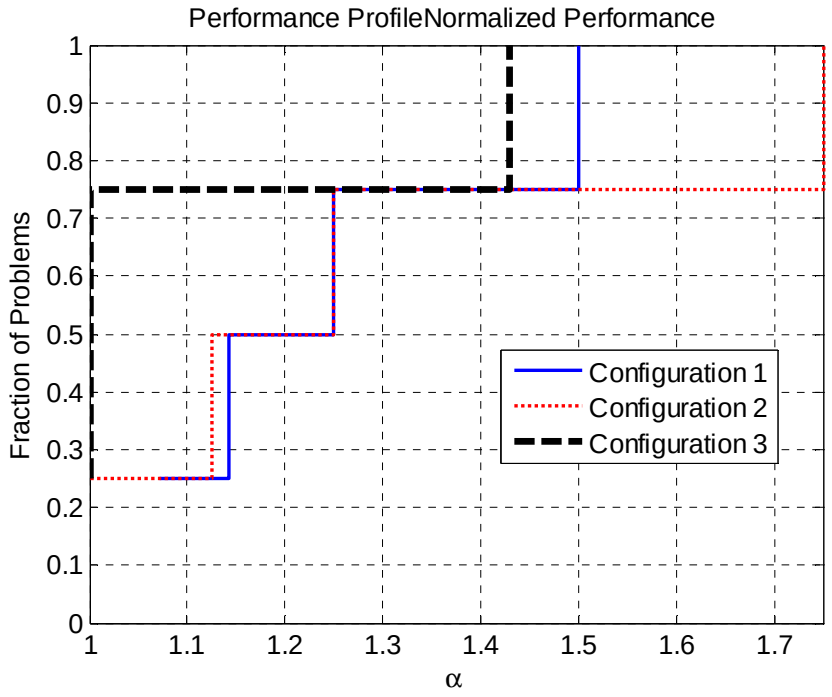


Figure 2.4: Performance profile for the illustrative example given in Table 2.3.

2.4 Software Optimization

The numerical factorization and triangular solution phases of the direct solver are optimized for an efficient solution of the structures on SMP multi-core processors. We usually avoid optimization at the code level, i.e., we do not perform low level optimizations such as loop unrolling. Instead, we try to employ the most efficient algorithms and data structures for the sparse direct solution of structures on multi-core processors. We also tune the parameters of the algorithms for the best performance on finite element problems.

A structured approach is followed in order to tune the solver package. The performance optimization methodology described by Tersteeg et al. [124] is used. The optimization is performed iteratively as shown in Figure 2.5. Here, each iteration designs, implements and verifies only one change to the code.

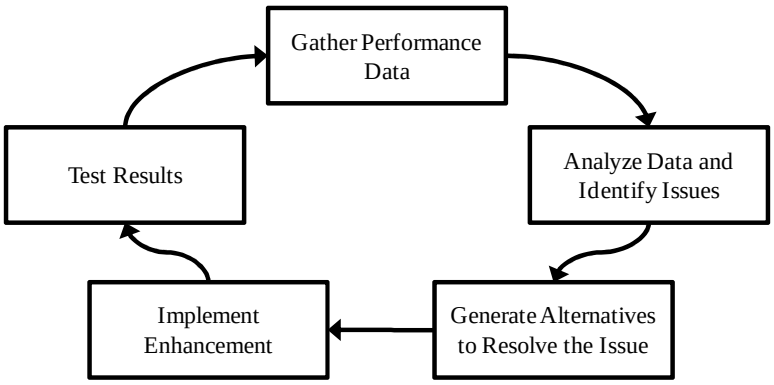


Figure 2.5: An iteration for the performance optimization [124]

A workload representing the realistic problems is required to measure performance and test the enhancements to the program. Tersteeg et al. [124] characterized the effective workloads as measurable, repeatable, static, and representative. For the tuning of the solver package, a large number of test structures are used. The test structures cover problems with different dimensions and node topology. Both 2D and 3D models composed of different element types are created for the test suite. See Section 2.6 and Appendix A for a detailed description of the test problems. The

performance data for the code is gathered using timers embedded with the code and AMD CodeAnalyst [125].

It is important to preserve the correctness of the solver after each attempt to increase the efficiency. A set of smaller structural model test problems are used to check the correctness of the solver package after each modification.

2.5 Implementation of the Solver Package

Sparse solver packages utilize sophisticated algorithms for the manipulation of the sparse data structures. However, the majority of the execution time is consumed in the core numerical factorization code, which is a small portion of the sparse solver code. For this reason, emphasis is placed on improving the performance of the core numerical factorization code. The complexity of the rest of the program is tackled using the object-oriented software design techniques. The design of the software addresses the following key considerations:

- High performance at the core factorization and solution routines.
- Flexibility in the preprocessing phase. The solver package allows experimenting with different preprocessing algorithms and different graph representations of structures.
- Clear distinction between the phases of the direct solver. This allows executing factorization and solution phases multiple times for a pivot-ordering found in the preprocessing phase.
- Measuring the execution time and memory requirements for the solution. Calculation of the performance criterion for a solution strategy.
- Portability and sustained performance at different platforms.
- Element connectivity and element matrices are taken as input to the solver package. The user is not required to build the stiffness matrix of the structure.

The stiffness matrix is assembled in an efficient fashion within the solver package.

- Use of external matrix-ordering libraries such as METIS, SCOTCH, CAMD, and etc.
- Implementation of hybrid factorization schemes.
- Configurable algorithms that allow experimenting with alternative values for the algorithm parameters.
- Support for command line interface for executing the solver within scripts.

The solver package is developed in C++, which supports objects and generic programming. Generic programming allows using abstraction without paying a performance penalty for virtual functions. Efficient and reusable mathematical libraries are developed using this technique such as the graph and matrix-ordering library developed by Lee et.al. [126]. C++ also allows developing performance critical components at a low level. For example, frequent dynamic memory allocation is avoided. Instead of dynamic allocation of small objects, total memory is allocated in large amounts and segments of the large memory block are used as necessary. The numerical factorization and solution is performed using tuned BLAS/LAPACK kernels (MKL [6]). The high performance can be sustained by the use of BLAS kernels tuned for a system.

The solver package is named as SES (**S**uper **E**lement **S**olver). The solver is composed of three main components:

1. Interface package for inputting the structural model and interaction with the solver package
2. Preprocessing package for finding pivot ordering
3. Solution package for performing factorization, condensation, forward elimination, and back substitution.

Figure 2.6 depicts the interaction between the packages. Each package uses the objects for storing the results. By merely relying on these objects, a direct link between

the packages is prevented. The preprocessing package returns an elimination tree object as the result of preprocessing. Then, an assembly tree object is built using the elimination tree, and the assembly tree is used for the symbolic and numerical factorization. In order to store the performance criteria for the solver package, a class that records the performance of different components is implemented. The class is designed as a singleton object, and any component of the solver can access the single instance of the class via a static interface.

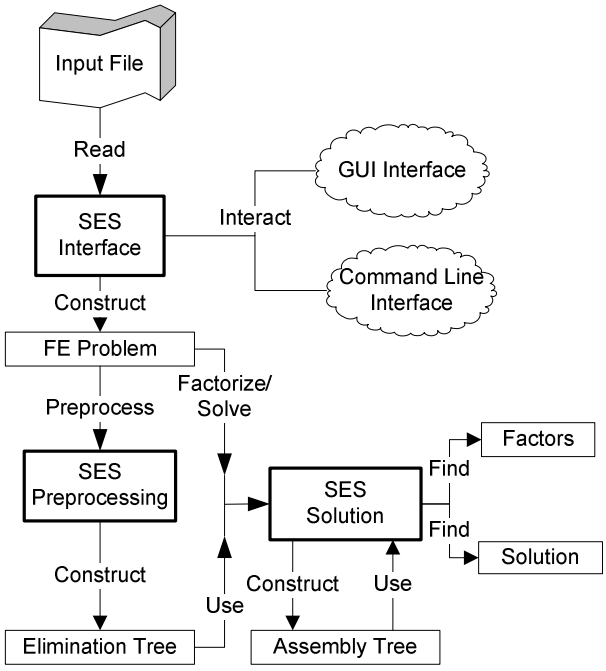


Figure 2.6: Main components of the SES solver package

Tasks performed by different components of the solver package are summarized as:

SES Interface

- Read the input structure from a data stream
- Apply an initial ordering for the nodes and elements of the structure
- Provide data streams to the solver package for outputting performance and program state

- Construct an efficient representation of the FE problem, which is used by SES preprocessing and solution packages
- Provide an interface for preprocessing and solution of problem with different strategies
- Provide an interface to modify element matrices and perform factorization for the same problem.

SES Preprocessing

- Fill-in reduction using local and hybrid ordering techniques
- Perform mesh coarsening.
- Construct the elimination tree, a tree storing the partial ordering of the elimination steps.
- Partition the structural model into a number of partitions.
- Record the execution time for the preprocessing algorithms.

SES Solution

- Perform symbolic factorization
- Build the assembly tree, a tree storing the assembly sequence of the elements.
- Find balanced workloads for parallel numerical factorization and solution.
- Find performance criterion for a given assembly tree.
- Perform symbolic factorization, condensation, and solution.
- Store statistics for symbolic factorization, condensation, and solution
- Assemble the element stiffness matrices
- Perform numerical factorization, condensation, and solution
- Record the execution time of factorization, condensation, and solution

2.6 Test Suites

A large number of test problems are used for evaluating the performance of the alternative solution strategies and for tuning the solver package. Both 2D and 3D

problems are generated for the experiments. The solver uses the node connectivity information (topology) of the structure and does not depend on finite element formulations. Therefore, generalized element types are used to construct the test problems. Each generalized element may represent different finite element formulations from various applications. The models are constructed using the following generalized element types:

1. 2D quadrilateral element with 4 nodes and 2 variables at each node
2. 2D frame elements with 2 nodes and 3 variables at each node
3. 3D solid elements with 4 nodes and 3 variables at each node
4. 3D frame elements with 2 nodes and 6 variables at each node

The number of variables at each node does not affect the pivot orderings of matrix ordering programs since a supervariable graph is used for the matrix ordering programs. However, the number of fill-ins and operation counts increase for a pivot ordering as the number of variables at each node increases. Therefore, small differences in the pivot orderings may have a larger impact on the operation count for models with a larger number of variables at each node.

In order to measure the numerical factorization times with satisfactory precision, the test problems should be large enough. Therefore, test problems are chosen to have factorization times greater than 0.05 seconds on the system where the numerical experiments are performed. Additionally, to prevent performance degradation due to the memory limitations, the number of entries in the lower diagonal factors is limited to $6.7 \cdot 10^8$ for 2D problems. $6.7 \cdot 10^8$ double precision entries approximately require 5 GB memory. For 3D problems, the limit for number of factors is $8 \cdot 10^8$, which is equivalent to 6 GB memory. These restrictions are for performing factorization using only the main memory (8 GB in the test setup).

There are 670 test problems having regular geometries with various aspect ratios. The model dimensions are summarized in Table 2.4. The number of elements in x, y and

z directions is given in this table. The models are generated for all possible unique combinations of the dimensions given in Table 2.4. As an example, 3D models with 10 elements in the x, y and z directions (10×10×10) are shown in Figure 2.7 for solid and frame elements. The detailed statistics for the uniform test problems can be found in Appendix A.

In addition to the test problems with regular geometries, test problems with irregular geometries are also used for the experiments. There are 86 irregular models that can be solved using only the main memory, which are composed of 21 problems with 2D quadrilateral elements, 21 problems with 2D frame elements, 22 problems with 3D solid elements, and 22 problems with 3D Frame elements. The properties of test problems with irregular geometries are also given in Appendix A.

Element Type	Number of Models	Model No.	Dimensions
2D quadrilateral	56	1-56	{20, 30, 40, 50, 60, 70, 80, 90}× {1000, 2000, 3000, 4000, 5000, 7500, 10000}
2D quadrilateral	27	57-83	{100, 200, 300, 400, 500, 1000}× {100, 200, 300, 400, 500, 1000, 1500}
2D frame	56	84-139	{20, 30, 40, 50, 60, 70, 80, 90}× {1000, 2000, 3000, 4000, 5000, 7500, 10000}
2D frame	27	140-166	{100, 200, 300, 400, 500, 1000}× {100, 200, 300, 400, 500, 1000, 1500}
3D solid	70	167-236	{5, 10, 15, 20}× {5, 10, 15, 20}× {75, 100, 125, 150, 175, 200, 250}
3D solid	119	237-355	{10, 15, 20, 25, 30, 35}× {10, 15, 20, 25, 30, 35}× {10, 15, 20, 25, 30, 35, 40, 45, 50}
3D solid	63	356-418	{25, 50, 75, 100, 125, 150}× {25, 50, 75, 100, 125, 150}× {4, 5, 6}
3D frame	70	419-488	{5, 10, 15, 20}× {5, 10, 15, 20}× {75, 100, 125, 150, 175, 200, 250}
3D frame	119	489-607	{10, 15, 20, 25, 30, 35}× {10, 15, 20, 25, 30, 35}× {10, 15, 20, 25, 30, 35, 40, 45, 50}
3D frame	63	608-670	{25, 50, 75, 100, 125, 150}× {25, 50, 75, 100, 125, 150}× {4, 5, 6}

Table 2.4: Test problems with regular geometries that can be solved using 8 Gbyte main memory only.

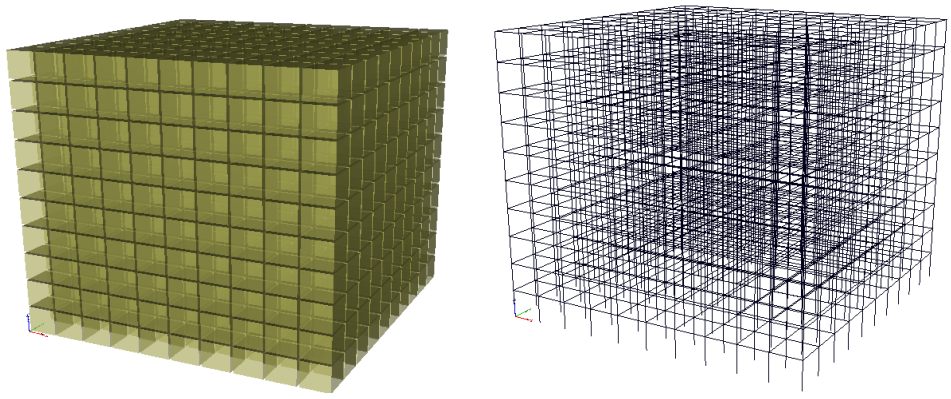


Figure 2.7: For 3D solid and 3D frame elements, 10×10×10 FE models with regular geometries

It’s not feasible to use all test problems for a large number of numerical experiments performed for tuning the solver package. For tuning the solver package, 40 regular test problems are selected among the 670 regular problems. Table 2.5 shows the selected problems. This test suite is referred to as benchmark suite of 40 test problems throughout this study.

2D quadrilateral	2D frame	3D solid	3D frame
1. q20×5000	11. f20×5000	21. s5×10×100	31. f5x10x100
2. q20×10000	12. f20×10000	22. s15×15×250	32. f15×15×250
3. q50×1000	13. f50×1000	23. s20×20×150	33. f20×20×150
4. q50×10000	14. f50×10000	24. s10×15×50	34. f10×15×50
5. q90×1000	15. f90×1000	25. s15×30×50	35. f15×30×50
6. q90×5000	16. f90×5000	26. s20×20×20	36. f20×20×20
7. q200×300	17. f200×300	27. s20×30×40	37. f20×30×40
8. q300×1000	18. f300×1000	28. s30×30×30	38. f30×30×30
9. q500×500	19. f500×500	29. s75×150×5	39. f75×150×5
10. q500×1500	20. f500×1500	30. s125×125×6	40. f125×125×6

Table 2.5: Benchmark suite of 40 test problems. The test suite is used to tune the solver package.

Once the solver is tuned, the performance of the solver package is demonstrated using 8 large test problems that were not used before. Figure 2.8 shows the 8 large test problems, which are composed of 2 models with 2D quadrilateral elements, 2 models with 2D frame elements, 2 models with solid elements, and 2 models with 3D frame elements. The problem sizes are chosen so that the 8 GB main memory is enough for an in-core factorization and triangular solution with 100 RHS vectors. Table 2.6 shows the statistics for the large test problems. Finally, the performance of the out-of-core solution is evaluated for 8 very large problems for which the solution cannot be performed using only the main memory. The geometry of the very large problems is similar to the ones shown in Figure 2.8. However, more elements are used for creating these problems. Table 2.7 shows the statistics for the very large test problems.

For all test problems, the variables at the bottom nodes are not active (support nodes), i.e., they do not contribute to the coefficient matrix. The rest of the nodes are

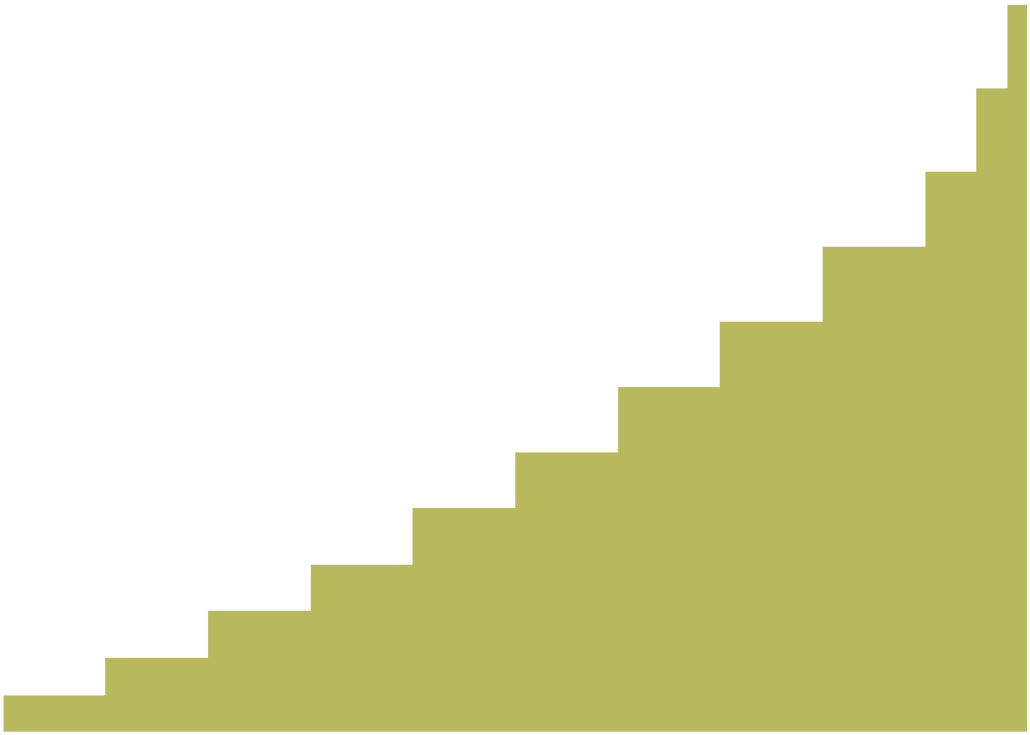
active. Due to the finite element discretization, the coefficient matrices of models are structurally symmetric. The coefficient matrix is numerically symmetric if the element matrices are also symmetric. In this study, it is assumed that the coefficient matrix is also numerically symmetric.

Model Name	Number of Dofs	Median Bandwidth	Non-zero for HMETIS	Flop for HMETIS
Q2DL1	1.49E+06	664	1.70E+08	1.25E+11
Q2DL2	1.93E+06	840	1.91E+08	1.18E+11
F2DL1	2.07E+06	301	1.83E+08	1.22E+11
F2DL2	2.06E+06	502	1.98E+08	1.80E+11
S3DL1	522,900	1353	5.05E+08	1.34E+12
S3DL2	883,560	1413	5.22E+08	1.03E+12
F3DL1	751,008	809	4.70E+08	1.31E+12
F3DL2	424,848	734	3.87E+08	1.65E+12

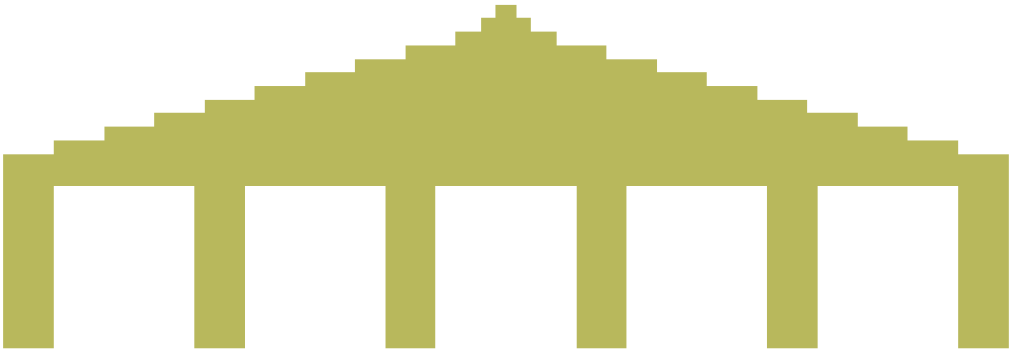
Table 2.6: Statistics for the benchmark suite with 8 large problems. Benchmark suite is used to evaluate the performance of the in-core solution.

Model Name	Number of Dofs	Median Bandwidth	Non-zero for HMETIS	Flop for HMETIS
Q2DVL1	4.83E+06	602	6.12E+08	7.32E+11
Q2DVL2	4.50E+06	602	4.94E+08	5.06E+11
F2DVL1	9.99E+06	301	9.24E+08	7.20E+11
F2DVL2	9.94E+06	502	1.01E+09	1.17E+12
S3DVL1	1.15E+06	3110	1.37E+09	5.66E+12
S3DVL2	2.21E+06	3135	1.56E+09	5.37E+12
F3DVL1	4.00E+06	969	2.30E+09	7.07E+12
F3DVL2	1.65E+06	361	1.35E+09	5.31E+12

Table 2.7: Statistics for the benchmark suite with 8 very large problems. Benchmark suite is used to evaluate the performance of the out-of-core solution.

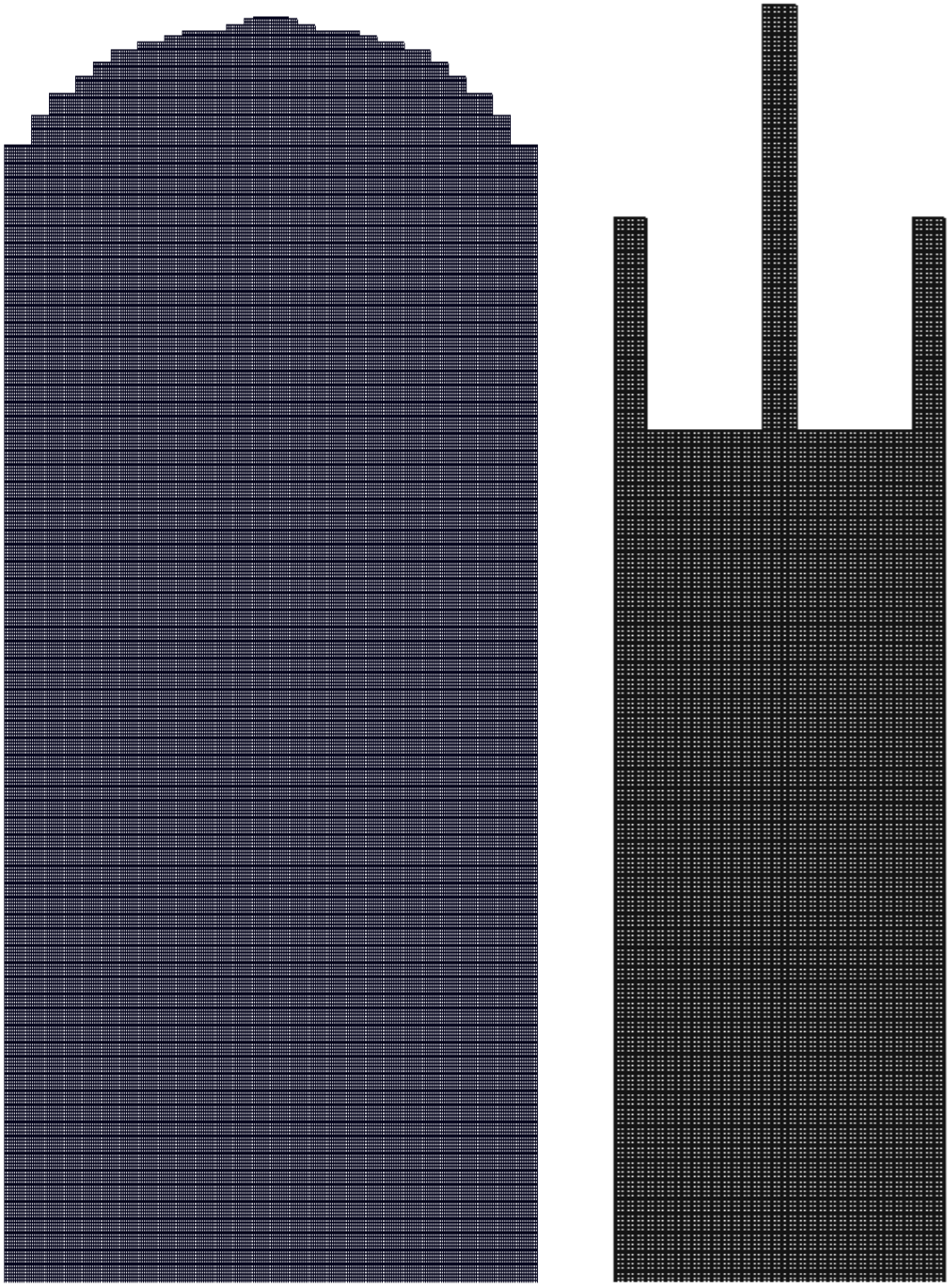


Q2DL1



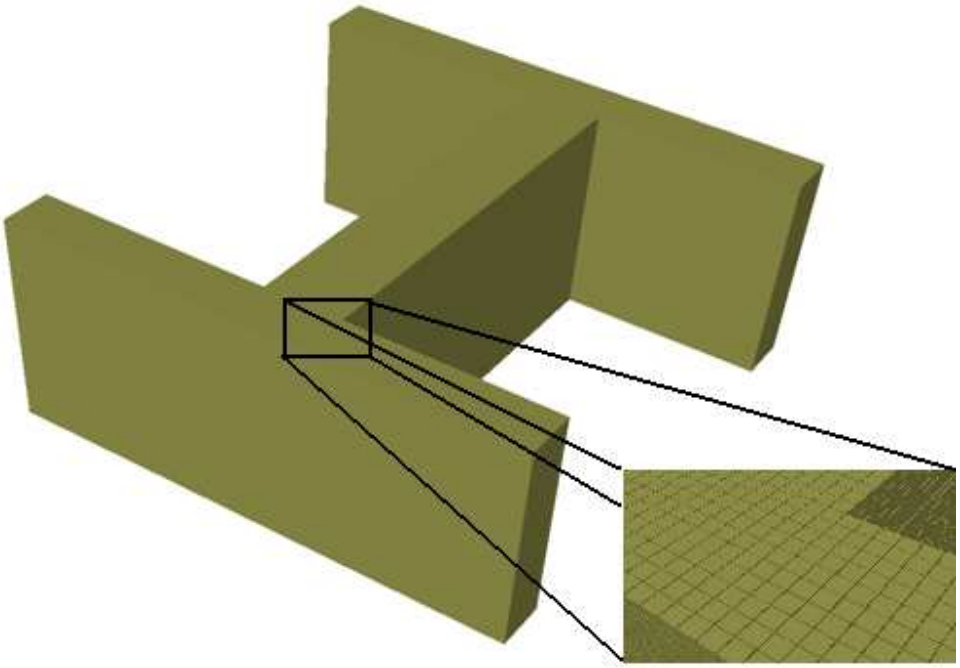
Q2DL2

Figure 2.8: Large test problems.

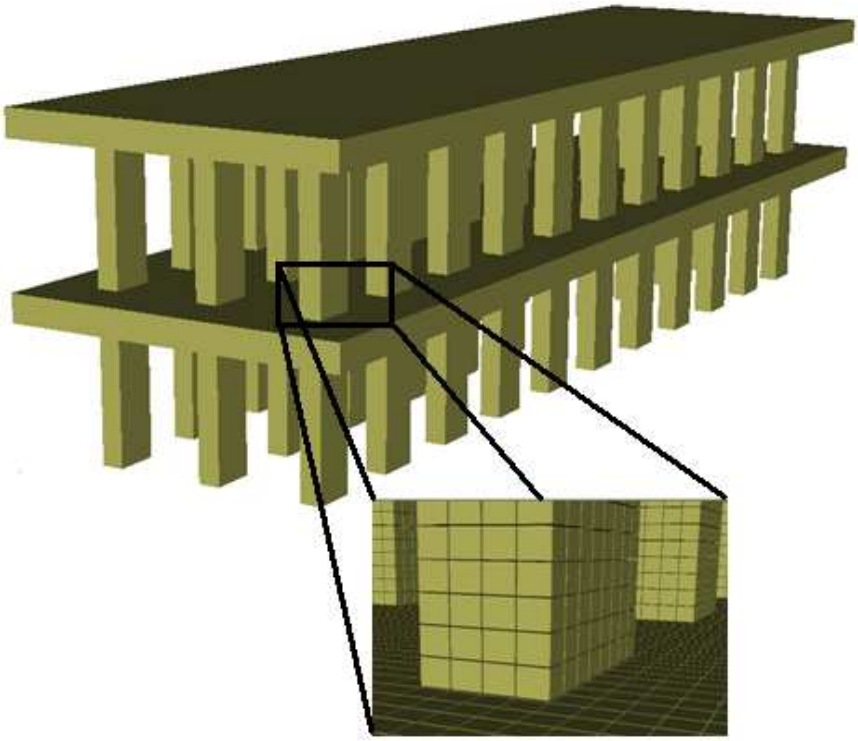


F2DL1 F2DL2

Figure 2.8 (cont.): Large test problems

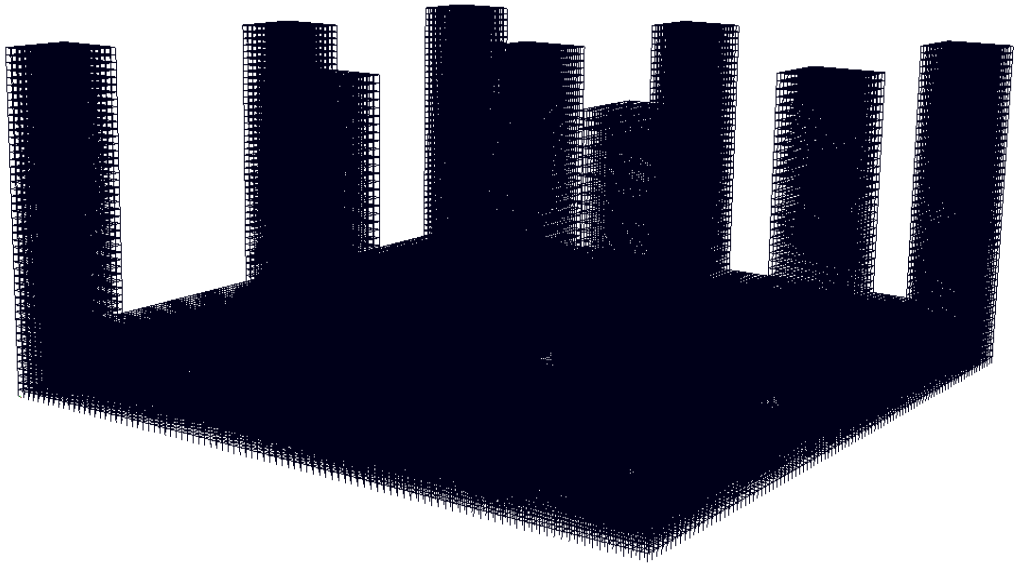


S3DL1

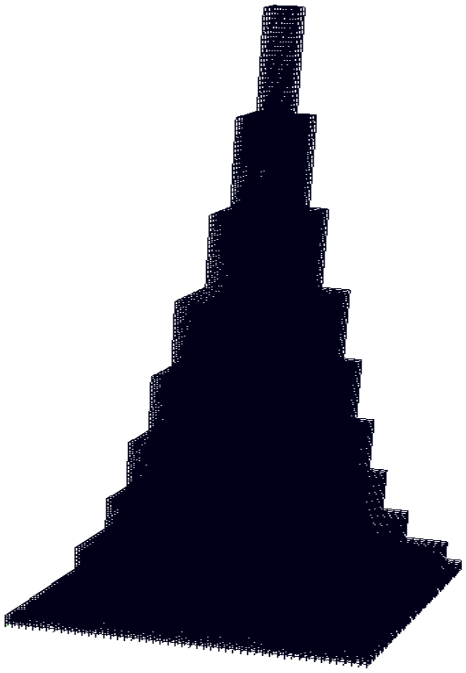


S3DL2

Figure 2.8 (cont.): Large test problems



F3DL1



F3DL2

Figure 2.8 (cont.): Large test problems

CHAPTER 3

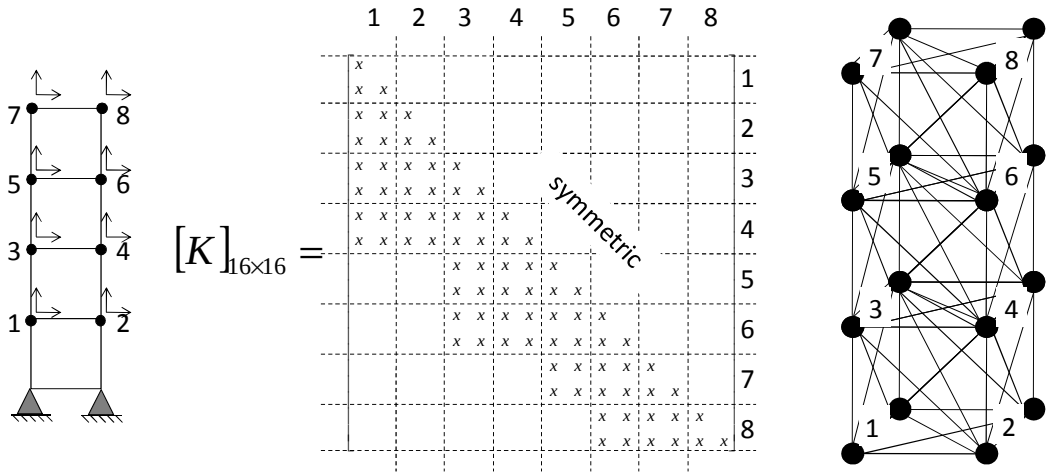
PREPROCESSING PHASE

The first phase of a general sparse solver is the preprocessing phase in which a pivot-ordering is determined that minimizes the execution time and memory requirements of the numerical factorization. A matrix ordering program can be used for this purpose. We also make use of the graph partitioning and mesh coarsening algorithms in the preprocessing phase. Our solver package unifies all these algorithms to find a pivot-ordering that minimizes the execution time of the numerical factorization. After a pivot-ordering is found, the preprocessing phase constructs an elimination tree [73].

3.1 Graph Representation of the Structures

The matrix ordering programs typically use a graph representing the nonzero structure of the sparse coefficient matrix. The coefficient matrix is the structural stiffness matrix, **K**, for the structural problems. In the graph representing the nonzero structure of the **K**, there is an edge between the graph vertices i and j if $\mathbf{K}_{i,j}$ is nonzero. As the factorization progresses, additional nonzero entries are introduced in **K** (fill-ins). The fill-ins introduce new edges between the vertices of the graph. The factorization can be modeled with a sequence of such graphs obtained by removing the vertex corresponding to the current pivot column and adding edges between all adjacent vertices of that vertex. Such graphs are called elimination graphs and they are useful for conceptual understanding of the fill-ins introduced during the factorization steps. An example of elimination graph is shown in Figure 3.1 for a simple structure which is also shown in this figure. As shown in Figure 3.1, the elimination graph has 16 vertices, the same number of vertices as the number of dofs in the example structure (or the number of columns of the stiffness matrix). The use of the elimination graph with a local ordering

algorithm is generally prohibitive since the size of the elimination graph grows in an unpredictable fashion as additional edges corresponding to the fill-ins are added to the graph. Instead of the elimination graph, a more compact representation called the quotient graph is more suitable for the local ordering algorithms [127]. A quotient graph stores the adjacency information by keeping the eliminated vertices in the graph to represent the edges added to their adjacent vertices. The elimination graph and quotient graph are identical in the beginning of the factorization, but unlike the elimination graph, in the later stages of the factorization, the quotient graph does not require a memory space larger than the memory required to store the original graph.



(a) **(b)** **(c)**

Figure 3.1: The graph representation of the stiffness matrix of the simple structure with 8 nodes. (a) the simple structure (b) stiffness matrix for the simple structure, (c) the elimination graph for the stiffness matrix. There are two dofs at each node other than the nodes with the supports.

For structural problems, the stiffness matrix columns corresponding to the dofs of a node typically have the same sparsity pattern. This is apparent in the example structure

shown in Figure 3.1. Here, the stiffness matrix entries for the dofs corresponding to a node are grouped into 2×2 dense matrix blocks. Instead of the graph shown in Figure 3.1, we can use a graph representing the nonzero pattern of 2×2 blocked stiffness matrix. In this compact graph representation, the blocks are represented with a weighted vertex where the weight of a vertex is equal to the number of dofs at a node and the edges represent the node connectivity information of the FE model. The size of the graph can be further reduced by compressing the nodes connected to the same set of elements into a single vertex in the graph. This compact representation of the coefficient matrix is called as supervariable graph [46, 128]. The matrix ordering programs run faster if a supervariable graph is used instead of the elimination graph since the supervariable graph is a compressed form of the elimination graph. Ashcraft [129] discussed the use of graph compression to reduce the execution time of the minimum degree orderings.

As an alternative to the supervariable graph, we can employ graphs representing the element connectivity of the structures for the partitioning algorithms. Several graph types have been used which have vertices for the elements and edges for the element connectivity information. An element communication graph [46, 130-131] connects the two elements if they share a node. A dual graph [47, 132] is a compact representation of the finite element mesh which has edges between the vertices only if k dimensional elements share $k-1$ dimensional boundaries. The vertices of the dual graph corresponding to elements with different dimension (i.e., frame element mixed with quadrilateral or solid elements) may be disconnected even though the elements share a node in the FE model. In this case, the element connectivity information cannot be represented correctly by the use of dual graph. The number of edges in an element communication graph is larger than the one in a dual graph for the same FE mesh. However, it provides a better representation for the fill-ins introduced during the factorization since there is an edge for each shared node of two adjacent elements. Additionally, unlike the dual graph, the connectivity information of the elements is never lost. Some other graph representations

and improvements to the dual graph are proposed by Kaveh and Rosta [130] and Topping and Ivanyi [133]. More information about the graph representations of finite element models can be found in References [46, 130-131, 134].

As an illustrative example, different graph representations of a structure are shown in Figure 3.2. Since there are typically more nodes than the elements in a structure, the supervariable graph is the largest among three alternative graph representations shown in Figure 3.2. As shown in Figure 3.2, the number of vertices for the element communication graph and dual graph is the same since the vertices represents the elements in the structure. However, the dual graph typically has fewer edges since there is an edge in the dual graph if only the connected elements share an edge in the finite element mesh.

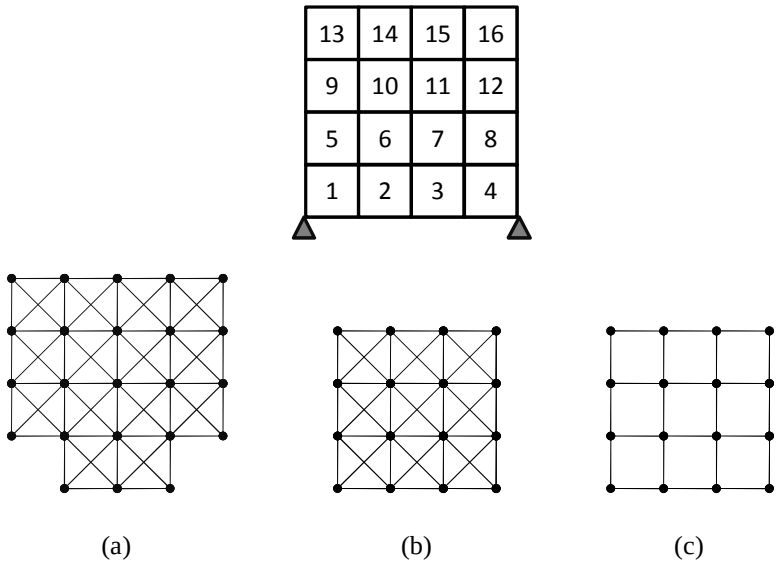


Figure 3.2 Graph representations of an example structure: (a) supervariable graph, (b) element communication graph, and (c) dual graph.

The SES solver package developed in this study allows using either supervariable or element communication graphs for the preprocessing algorithms. A dual graph is not implemented due to its limitations described previously. The matrix-ordering programs

that can be used in the SES solver package typically work with a supervariable graph. Partitioning and mesh coarsening algorithms described in the subsequent chapters can be used with either a supervariable or an element communication graph representing a structure (or a substructure).

3.2 Initial Node Numbering

The local matrix ordering algorithms choose the next pivot-column in a greedy fashion based on a heuristic function value such as minimum degree or minimum fill-in. There may be multiple candidate columns with the same heuristic function value at a stage of a local matrix ordering algorithm. The tie-breaking strategy for the multiple pivot candidates affects the quality of the ordering. A minimum degree algorithm typically chooses the next pivot according to the initial numbering of the columns if there is a tie between the candidate pivots. Therefore, the initial numbering of the graph vertices affects the quality of the pivot-orderings produced by a minimum degree ordering algorithm. George and Liu [60] showed that the two initial node numberings, the numbering of nodes based on the coordinate information and reverse Cuthill-McKee ordering, yield favorable fill-ins for the minimum degree ordering of an example square grid. These initial node numberings typically produced pivot-orderings with fewer fill-ins compared to the random node permutations.

The performance of various matrix ordering programs are evaluated using benchmark suites with FE test problems. In performance evaluations, the effect of initial node numbering can be eliminated by using the median result among several pivot-orderings found for different random node permutations. However, as stated previously, random node permutations may result in pivot-orderings yielding larger fill-ins compared to the original node numbering, especially for the local matrix ordering programs [63]. In the SES solver package, three types of initial node numberings are implemented:

- Random permutations – the nodes are shuffled randomly based on a random number generator with a seed value given by the user.
- Coordinate based node numbering – the nodes are ordered according to the ascending values of the x, y, and z coordinates. Here, first the nodes with the minimum y and z coordinates are ordered in ascending values for the x coordinates. Next, the y coordinate is incremented and the ordering according to the x coordinates is repeated. Finally, the z coordinate is incremented and the ordering is repeated for the nodes with second smallest z values. The coordinate based numbering is illustrated in Figure 3.3 for an example structure.
- Cuthill-McKee (CMK) and reverse CMK orderings – the nodes are ordered by using the CMK subroutine in the BOOST graph library [135].
- Node numbering in the input structure – the original numbering of the nodes in the input structure is used.

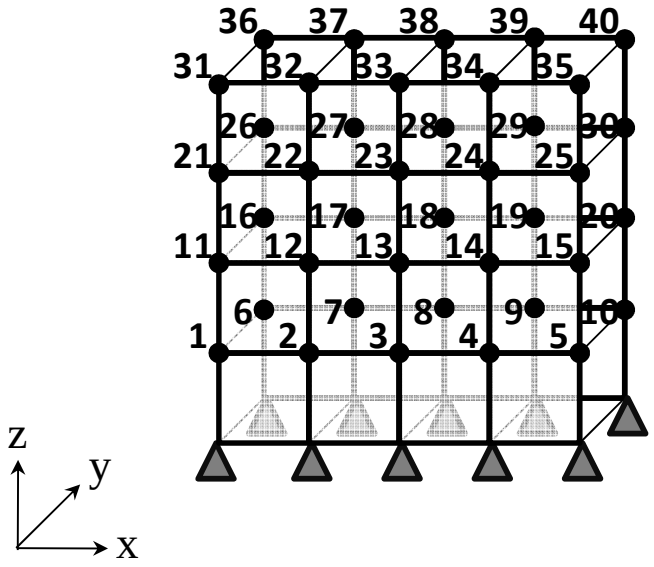


Figure 3.3: The initial numbering of the nodes based on the coordinate information of an example structure.

The initial node numbering algorithms reassign the numbers to the nodes in the input structure. The preprocessing algorithms are executed for the graph with the nodes numbered according to one of the initial node numberings described above. The effect of initial node numbering is illustrated later in Chapter 6.

3.3 Matrix Ordering Programs

There are several public domain matrix ordering programs that implement local heuristics and hybrid orderings. The SES solver package allows using the following matrix ordering programs developed by other researchers:

- AMF – Approximate minimum fill-in ordering in SCOTCH library [121]
- MMD – Multiple minimum-degree ordering in METIS library [69]
- BOOST-MMD – Multiple minimum-degree ordering in BOOST libraries [135]
- AMD – Approximate minimum degree ordering [63]
- HAMF – Hybrid ordering in SCOTCH library [121]
- HMETIS – Hybrid ordering in METIS library [69]

The first four matrix ordering programs shown above are based on the local heuristics. The last two programs, HAMF and HMETIS, are hybrid matrix ordering subroutines in the SCOTCH and METIS libraries respectively. The SES package allows experimenting with alternative parameters for the matrix-ordering programs. In this section, matrix-ordering programs are briefly explained and their adjustable parameters are discussed.

Performing factorization operations on a small number of pivot columns is not as fast as performing factorization on larger column blocks. Therefore, the pivot columns are coalesced (node amalgamation) in order to increase the size of the block sizes on which the factorization is performed [57]. The SCOTCH library can perform the node amalgamation within the matrix ordering programs. Namely, the minimum size of the

supernodes found with AMF can be set by the user. The parameter *cmin* determines the minimum size of the supernodes. If a supernode is smaller than *cmin*, it is merged with its parent in the elimination tree. The amalgamation of the supernodes typically introduces fictitious fill-ins in the coefficient matrix and the number of fictitious fill-ins can be controlled by the *fratio* parameter in AMF. The *fratio* determines the upper limit for the fill-in ratio with respect to the non-zero entries in the pivot columns. If the fill-in ratio is larger than *fratio*, then the node amalgamation is not performed even if a supernode is smaller than *cmin*. One of the advantages of node amalgamation is the increased efficiency of the BLAS3 kernels since the BLAS3 kernels run faster as the size of the dense matrix increases. Node amalgamation also reduces the size of the assembly tree and the number of update operations for the multifrontal method.

The SCOTCH library has an option to compress the input graph. The compressed graph is used if the ratio of the size of the compressed graph to the size of the original graph is below the parameter *cratio*. The graph compression can speed up the matrix-ordering programs.

MMD algorithms allow simultaneous elimination of the independent vertices in the elimination graph. The independent node elimination is controlled by the parameter *delta*. The parameter *delta* is the degree difference between the node(s) with the minimum degree and other nodes that can be eliminated simultaneously. Multiple eliminations of the nodes typically reduce the execution time of the minimum degree ordering. Local heuristics AMD and AMF do not allow multiple eliminations. MMD and BOOST-MMD are essentially the same ordering programs with different implementations. BOOST-MMD is implemented by using the C++ templates of the BOOST graph library. MMD typically executes faster than BOOST MMD since it is a low-level C program.

Hybrid matrix ordering HAMF is the default matrix ordering strategy in the SCOTCH library. In HAMF, nested dissections are performed until the partitions are

small enough. Then, the partitions are ordered using an approximate minimum fill-in ordering [70]. HAMF allows setting the stopping criteria for the nested dissections. If a partition found with the nested dissections has number of nodes smaller than the *vertnum* parameter, the partitioning is stopped and the partition is ordered with the local ordering. A distinct feature of the HAMF is that the exact connectivity information of the graph nodes at the boundaries is used for the local ordering of the partitions. The use of exact information typically increases the quality of the hybrid ordering [70]. As previously explained, the parameters for the AMF determine the features of the local ordering used for the partitions. It should be noted that SCOTCH library also allows using an AMD ordering for the local ordering of the partitions. The SES solver package allows the use of both AMF and AMD for the ordering of the partitioned graphs. Nevertheless, AMF typically yields fewer fill-ins at a cost of increased execution time.

Hybrid ordering HMETIS is similar to HAMF except that the MMD algorithm is used for local ordering of the partitions instead of AMF. The options for HMETIS allow controlling the matching algorithm, refinement algorithm, initial partitioning algorithm, and number of partitions at each step of nested dissection. Typically, default options of the HMETIS give satisfactory results. For the hybrid ordering HMETIS, the SES solver package allows using either a compressed graph (METIS_NodeWND subroutine in METIS [69]) or an uncompressed graph (METIS_NodeND subroutine in METIS [69]). For the METIS_NodeWND subroutine, it is assumed that the graph is already compressed and the graph vertices have weights representing the number of variables at each vertex. On the other hand, for the METIS_NodeND subroutine, the compression is applied within the METIS package.

A supervariable graph is employed for all matrix ordering programs implemented in the SES solver package. Although the SCOTCH library allows using a mesh representation of the structure, this feature has not yet been implemented. George [60] stated that the use of a mesh representation for the finite element problems can reduce the

memory requirements of ordering algorithms, especially for the meshes with higher-order elements.

3.4 Graph Partitioning

Graph partitioning can be used to increase the concurrency of the sparse direct solution. If a hybrid matrix-ordering program is used for fill-in reduction, the graph partitioning is performed implicitly to find disconnected graph vertices corresponding to the pivot columns that does not introduce fill-in to each other. Therefore, the elimination tree produced by a hybrid ordering program is suitable for exploiting tree-level parallelism [74, 77]. To illustrate this point, Figure 3.4 shows the partitions found with HMETIS hybrid ordering for a 2D model. As shown in Figure 3.4, HMETIS finds 64 partitions for which the factorization steps can be performed in parallel.

Graph partitioning can also be performed at a structural level to harness parallel processing in the pre-processing and post-processing stages. An element communication graph can be used to partition the structure into independent components. In order to obtain partitions with balanced workloads, node weights of the element communication graph should be a close approximation to the computational work related to an element. As an alternative to the element communication graph, a supervariable graph can be used for partitioning of the structure. However, this may result in duplication of boundary elements in neighbouring partitions.

The SES solver package allows using partitioning algorithms from the METIS graph partitioning library. SES provides interfaces for k-way partitioning and recursive nested dissection in METIS. Either an element communication or a supervariable graph representation of a structure can be used for partitioning. If a supervariable graph is used, the boundary elements are assigned to a partition according to partitioning of the element nodes. Namely, an element is assigned to the partition that has the majority of element nodes.

Some preliminary experiments are performed to investigate the efficiency of explicit partitioning for the numerical factorization. Figure 3.5 shows the 64 partitions found with METIS by using the element communication graph. Similarly, Figure 3.6 shows the 64 partitions found by using the supervariable graph. If Figure 3.5 and Figure 3.6 are compared with Figure 3.4, we see that METIS partitioning subroutines creates irregular boundaries between the partitions compared to the partition boundaries found in the HMETIS hybrid ordering. Consequently, the number of nonzero entries in the factors is larger than the one found with HMETIS hybrid ordering for the explicit partitioning of the structures (non-zeros are given below the figures). The difference is mainly due to the different goals of the partitioning and matrix ordering algorithms. The main goal of a partitioning algorithm is to find balanced partitions. This may not produce the minimum non-zero since minimizing the edge cuts is more important than the balanced partitions for fill-in minimization. Bruce et al. [136] also stated the importance of allowing some imbalance between the partitions for minimizing the fill-ins.

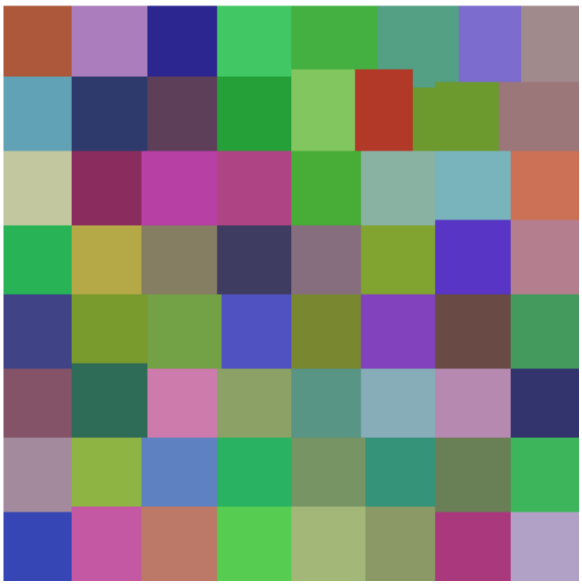


Figure 3.4: Partitions found with the HMETIS hybrid ordering for q100×100. Non-zero = 1.285E6 for the hybrid ordering with the partitions illustrated above.

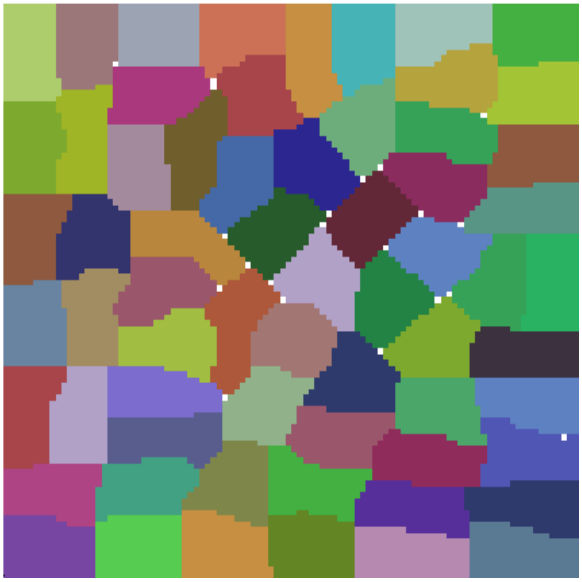


Figure 3.5: 64 partitions found with METIS recursive nested dissections on the element communication graph for q100×100. Non-zero = 1.673E6 after ordering partitions and separators with AMF.

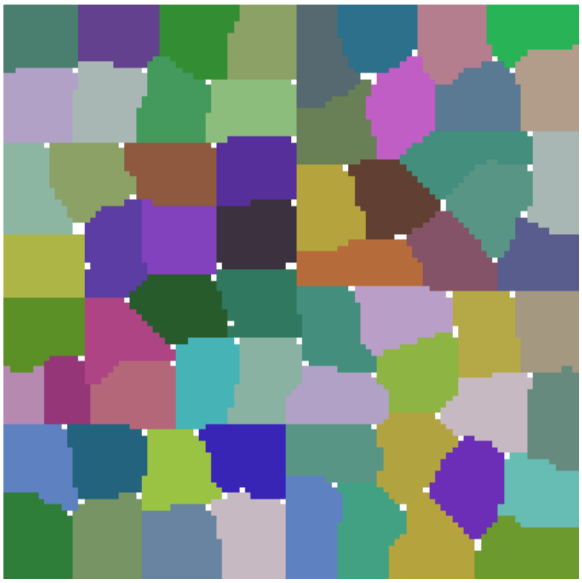


Figure 3.6: 64 partitions found with METIS recursive nested dissections on the supervariable graph for q100×100. Non-zero = 1.579E6 after ordering partitions and separators with AMF.

As illustrated in the previous examples, the application of explicit partitioning may increase the number of non-zeros compared to the implicit partitioning of the hybrid ordering programs. The main goal for performing explicit partitioning is to increase the degree of parallelism. However, the pivot orderings found with hybrid matrix ordering programs are already suitable for parallel processing. Therefore, partitioning may not be essential for high-performance in parallel factorization. Table 3.1 shows the parallel factorization performance for preprocessing q500×500 test problem with and without partitioning. There are four cores in the test machine, therefore the FE mesh is partitioned into four independent components. A thread is spawned for each core and the factorization of the partitions is performed simultaneously on four cores. As shown in Table 3.1, partitioning the FE mesh with METIS package increases the factorization flop. Therefore, both parallel and serial factorization times are increased compared to no partitioning. We further investigate the efficiency of the partitioning algorithms for a benchmark suite.

Preprocessing Algorithms	**Non-zero (10^7)**	**Factorization Flop (GFlop)**	**Serial Factorization Time (sec)**	**Four-Thread Factorization Time (sec)**
AMF	5.55	23.07	4.96	1.84
METIS	5.07	22.47	4.82	1.48
4 Partitions + AMF	7.50	57.07	10.98	4.04
4 Partitions + METIS	6.32	33.92	6.64	2.09

Table 3.1: Effect of explicit partitioning on the numerical factorization times for the problem q500×500.

3.5 Mesh Coarsening

Guney and Will [134] proposed a solution scheme that uses a structural model comprising super-elements created by merging adjacent elements in the original structural model. The new super-elements created by the coarsening scheme are larger than the original finite elements (the super-elements have more nodes compared to the original elements) and the graph representation of the coarsened mesh, namely, the supervariable or element communication graphs, is typically smaller than the size of the graph representation of the original mesh. Therefore, the preprocessing phase executes substantially faster if a coarser model is used instead of the original model. The coarsening scheme also improves the efficiency of the multifrontal factorization. The effect of coarsening is similar to node amalgamation in the multifrontal method [57]. The coarsening enlarges the frontal matrices at the leaves of the assembly tree since super-elements are larger than the original finite elements used to construct the assembly tree for the original mesh. Similarly, the number of assembly tree nodes are reduced since the number of super elements at the coarsened mesh is smaller than the number of original finite elements. Therefore, the number of update operations performed for assembly tree nodes is reduced if a coarsened mesh is employed for the numerical solution. Additionally, the BLAS3 kernels typically run faster since the frontal matrix sizes are large compared to the frontal matrices for the original finite element mesh.

In this study, two alternative coarsening algorithms are implemented. The first is the adaptation of the coarsening algorithm originally proposed by Guney and Will [134]. Here, the original elements are merged with their adjacent elements. The steps of the element based coarsening algorithm are given as follows:

1. Visit elements in the increasing order of their x, y and z coordinate. Let’s call currently visited element as *e*.

2. If *e* and its adjacent elements, *Adj*(*e*), are not already coalesced with some other element then *e* is eligible for coarsening, go to step 3. Otherwise *e* is not eligible, go to step 1.
3. Merge *e* and all of its adjacent elements to form super-element $E = \{e \cup Adj(e)\}$.
4. Repeat steps 2 to 4 until all original elements are visited.

Figure 3.7 illustrates the steps of coarsening on an example problem. The first element that is eligible for merging with its adjacency is element 1. For *e* representing element 1, *Adj*(*e*) has elements 2, 5 and 6 and the coarsened element *E* is the union of elements 1, 2, 5 and 6. As shown in Figure 3.7, *Adj*(*E*) are marked so that when these elements are visited later in Step 1, the criterion given at Step 2 can be checked easily. When the coarsening is completed, there are 4 super-elements in the coarsened mesh compared to the 16 elements in the original finite element mesh.

The main disadvantage of element based coarsening is that we do not have direct control on number of nodes eliminated at super-elements formed by coarsening of the finite elements. An alternative to element based coarsening is coarsening based on the node connectivity information. This is similar to the element based coarsening. However, the elements are now merged by first finding the nodes that will be eliminated at each super-element. This allows choosing the number of nodes eliminated at super-elements. The steps in the node based coarsening algorithm are given as follows:

1. Visit nodes in the increasing order of their x, y and z coordinate. Let's call currently visited node as *n*.
2. If *n* is not already coalesced with some other node, then *n* is eligible for coarsening, go to step 3. Otherwise, *n* is not eligible, go to step 1.
3. Merge all elements connected to *n* to form super-element, *E*, with the nodes $\{n \cup Adj(n)\}$.
4. Repeat steps 2 to 4 until all original nodes are visited.

The steps of node based coarsening are depicted in Figure 3.8 for the same example mesh used to demonstrate the element based coarsening in Figure 3.7. Here, the coarsening algorithm first picks the node with the smallest x, y and z coordinates. Then, it merges all elements connected to this node (elements 1 and 5). The newly formed super-element contains the eligible node and its adjacent nodes. The nodes of the newly formed super-element are marked as not eligible. This allows easy determination of the node eligibility at the Step 2 of the node based coarsening algorithm given above.

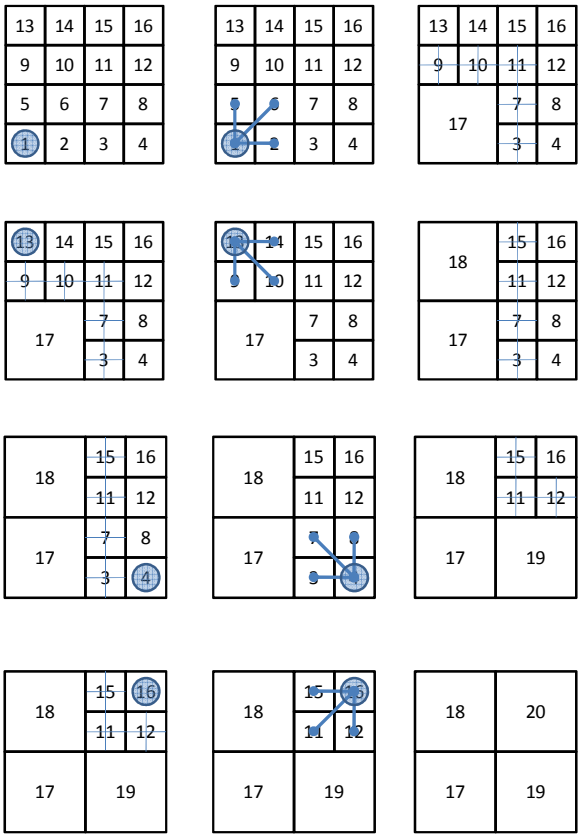


Figure 3.7: Element based coarsening for a sample 5×5 mesh. Each row in the figure illustrates selecting an eligible element and merging it with its adjacent elements.

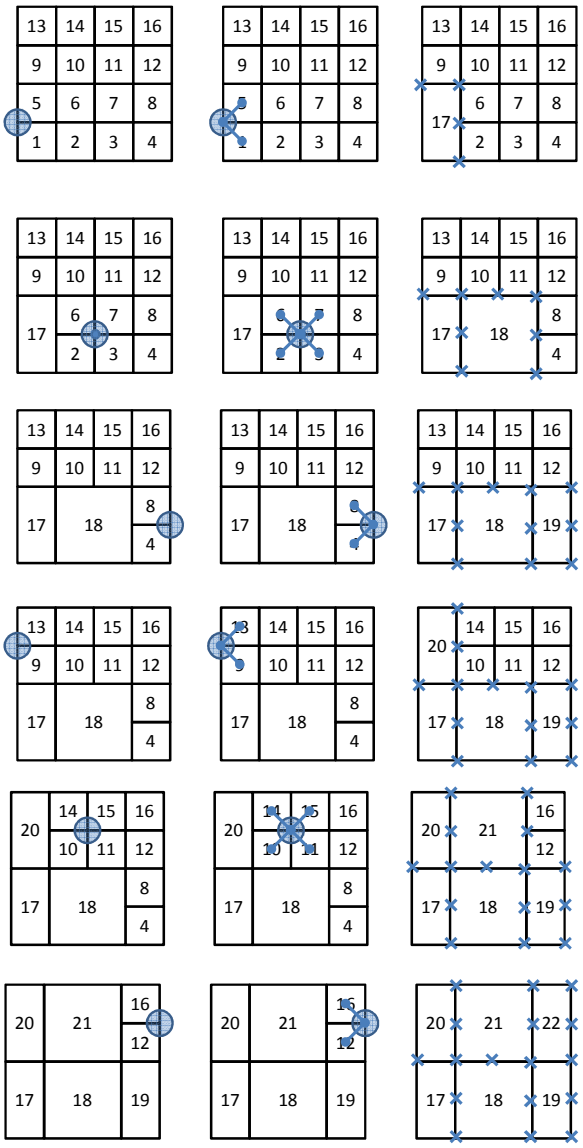


Figure 3.8: Node based coarsening for a sample 5×5 mesh. There is a node at each corner of an element and bottom nodes are fully restrained. Each row in the figure illustrates selecting an eligible node and merging the elements connected to it.

The coarsening schemes illustrated in Figure 3.7 and Figure 3.8 use a single eligible element and node respectively at each step of the coarsening. Aggressive coarsening schemes can be employed by using multiple eligible elements/nodes at each coarsening step. The *eleco* parameter of the SES solver package determines the number of eligible elements selected at each element based coarsening step. Figure 3.9 shows the

coarsened mesh obtained for different *eleco* values for an example 50×50 quadrilateral mesh. As shown in Figure 3.9, a coarser mesh is obtained as we increase the *eleco* value. Similarly, the *nodeco* parameter determines the number nodes eliminated at each super-element for a node based coarsening algorithm. Figure 3.10 shows the coarsened mesh for different *nodeco* values. As shown in Figure 3.10, coarser meshes are obtained as we increase the *nodeco* value. In Figure 3.10, the white spots for the *nodeco* = 8 show the elements that are not merged with any elements.

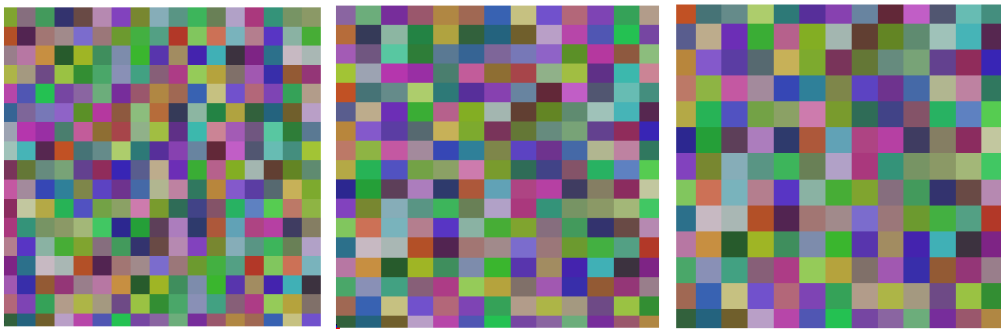


Figure 3.9: The element based coarsening for *eleco*=1, 2, and 4 from left to right respectively. The original model is q50×50. Each super-element in the coarsened mesh is painted with a different color.

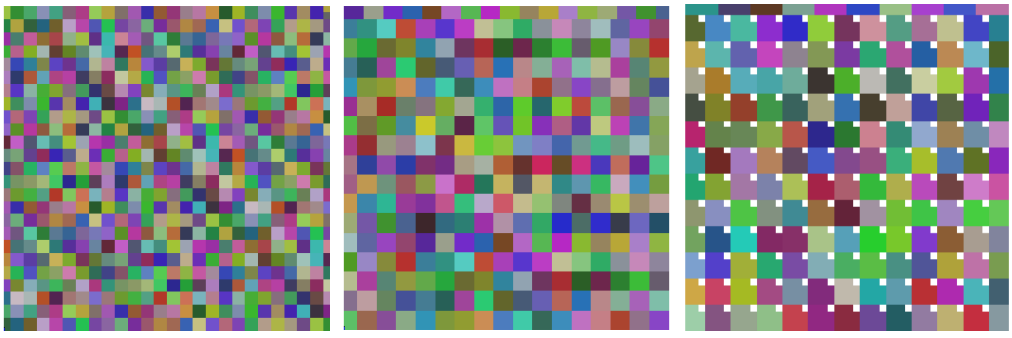


Figure 3.10: The node based coarsening for *nodeco*=1, 4, and 8 from left to right respectively. The original model is q50×50. Each super-element in the coarsened mesh is painted with a different color.

Next, we demonstrate the effect of the coarsening on four example test problems. Pivot-orderings are found with HMETIS for the numerical experiments in this section. For the test problem q500×500, Table 3.2 and Table 3.3 show the effect of coarsening strategies on different stages of the solver package for element based coarsening and node based coarsening schemes respectively. As shown in Table 3.2 and Table 3.3, both coarsening algorithms can greatly reduce the matrix-ordering time, about three times faster than the one for the original mesh. Moreover, analysis and factorization times are reduced significantly. However, the number of eliminated nodes at super-elements should be carefully chosen for node based coarsening algorithms since the coarsening may significantly increase the factorization flop and time. For example, the factorization flop and time for *nodeco* = 3 in Table 3.3 are significantly larger than the ones for the original mesh. For problem q500×500, *nodeco* = 4 and 8 give the best factorization times for node based coarsening. For element based coarsening, the best performance is obtained if a single element is merged with its adjacency at each coarsening step.

For the test problem f500×500, Table 3.4 and Table 3.5 show the effect of coarsening strategies for the element and node based coarsening schemes respectively. The element based coarsening does not improve the performance of the numerical factorization for this problem. Whereas, the node based coarsening can reduce the matrix ordering, analysis, and factorization times.

Compared to the performance on 2D problems, the improvements due to the coarsening scheme are less significant for 3D problems. Table 3.6 and Table 3.7 show the performance of coarsening strategies for s15×15×250 for element and node based coarsening schemes respectively. While some improvement can be observed in matrix ordering and analysis times for the element based coarsening scheme, the benefits of the coarsening are offset by the increase in the factorization times due to the increase in the flop for factorization. As shown in Table 3.7, the node based coarsening scheme also increases the factorization times for the s15×15×250 test problem.

Table 3.8 and Table 3.9 show the performance of coarsening strategies for f20×20×20 for element and node based coarsening schemes respectively. For this problem, the element based coarsening scheme is not beneficial for reducing the factorization time, whereas, the node based coarsening with single eliminated node at each super-element reduces the factorization time.

The numerical experiments illustrate that benefits of coarsening are different for problems with different dimensionality. For 2D problems, a more aggressive coarsening can be applied with larger number of elements eliminated at each super-element. On the other hand, for 3D problems, the factorization flop grows dramatically if an aggressive coarsening scheme is employed, which merges a large number of elements to form super-elements. A coarsening scheme that eliminates a single node at each super-element usually gives the most favorable results for 3D problems. Further numerical experiments are performed in Chapter 6 in order to determine the effect of *eleco* and *nodeco* parameters on the execution time of the solver package.

eleco	Matrix Ordering Time (sec)	Analysis Time (sec)	Factorization Time (sec)	Number of Tree Nodes	Flop (10^9) for Factorization
0	1.91	3.99	6.36	130,823	21.64
1	0.70	1.70	5.19	55,693	20.86
2	0.54	0.86	5.68	19,232	28.13
3	0.55	0.77	5.7	17,872	29.04

Table 3.2: Performance of the element based coarsening scheme for q500×500.

nodeco	Matrix Ordering Time (sec)	Analysis Time (sec)	Factorization Time (sec)	Number of Tree Nodes	Flop (10^9) for Factorization
1	1.67	3.48	5.87	125,336	21.32
2	1.13	2.41	5.88	83,342	24.05
3	1.25	2.23	9.18	71,145	46.33
4	0.69	1.69	5.26	55,673	22
5	0.96	1.83	6.79	58,613	30.77
6	0.65	1.43	6.28	41,845	28.45
7	0.93	1.35	8.99	36,675	45.97
8	0.56	1.46	5.37	45,744	22.27
9	0.41	0.72	5.85	15,789	28.88

Table 3.3: Performance of the node based coarsening scheme for q500×500.

eleco	Matrix Ordering Time (sec)	Analysis Time (sec)	Factorization Time (sec)	Number of Tree Nodes	Flop (109) for Factorization
0	1.39	4.54	11.42	192,829	46.05
1	0.97	3.07	12.80	112,949	63.37
2	1.06	1.44	11.34	33,424	60.45
3	1.1	1.38	11.7	29,894	62.89

Table 3.4: Performance of the element based coarsening scheme for f500×500.

nodeco	Matrix Ordering Time (sec)	Analysis Time (sec)	Factorization Time (sec)	Number of Tree Nodes	Flop (10^9) for Factorization
1	0.80	3.96	10.89	191,559	50.33
2	1.01	2.92	10.16	111,416	47.62
3	1.01	2.38	11.31	84,669	55.73
4	1.12	2.18	12.4	70,339	63.68
5	0.88	1.96	9.65	64,990	47.65
6	0.76	1.54	10.16	51,009	50.14
7	1.02	1.6	12.29	48,360	63.15
8	0.37	1.23	9.2	40,168	45.17
9	0.63	1.02	10.5	21,240	55.33

Table 3.5: Performance of the node based coarsening scheme for f500×500.

eleco	Matrix Ordering Time (sec)	Analysis Time (sec)	Factorization Time (sec)	Number of Tree Nodes	Flop (10^9) for Factorization
0	0.8	1.08	24.29	19,050	142.99
1	0.51	0.5	29.24	8,307	179.09
2	0.29	0.29	30.66	3,968	191.96
3	0.27	0.25	28.1	3,472	176.29

Table 3.6: Performance of the element based coarsening scheme for s15×15×250.

nodeco	Matrix Ordering Time (sec)	Analysis Time (sec)	Factorization Time (sec)	Number of Tree Nodes	Flop (10^9) for Factorization
1	1.10	0.87	25.71	15,977	156.23
2	1.01	0.7	29.89	11,462	180.21
3	1.40	0.79	61.38	10,112	384.21
4	1.37	0.73	67.74	8,786	429.46
5	1.54	0.7	71.68	8,234	449.89
6	1.42	0.64	70.93	7,509	445.26
7	1.51	0.61	68.42	7,040	425.49
8	1.38	0.57	69.45	6,160	439.02
9	1.33	0.36	69.68	3,379	451.52

Table 3.7: Performance of the node based coarsening scheme for s15×15×250.

eleco	Matrix Ordering Time (sec)	Analysis Time (sec)	Factorization Time (sec)	Number of Tree Nodes	Flop (10^9) for Factorization
0	0.06	0.2	7.32	6,014	40.38
1	0.06	0.16	10.17	3,656	62.57
2	0.06	0.1	11.73	1,445	73.79
3	0.06	0.09	12.67	1,275	78.43

Table 3.8: Performance of the element based coarsening scheme for f20×20×20.

nodeco	Matrix Ordering Time (sec)	Analysis Time (sec)	Factorization Time (sec)	Number of Tree Nodes	Flop (10^9) for Factorization
1	0.04	0.19	6.23	6,057	35.47
2	0.06	0.15	10.47	3,475	65.94
3	0.06	0.14	12.2	2,745	77.17
4	0.06	0.13	12	2,415	74.69
5	0.07	0.12	11.61	2,283	70.67
6	0.07	0.11	11.91	2,069	73.35
7	0.07	0.11	11.61	1,841	70.71
8	0.08	0.11	12.72	1,663	73.82
9	0.07	0.08	12.55	1035	77.27

Table 3.9: Performance of the node based coarsening scheme for f20×20×20.

3.6 Object Oriented Design of the Preprocessing Phase

The SES solver package allows the use of coarsening, matrix-ordering, and partitioning algorithms in any order for different regions of the structure. This flexibility is achieved by an object oriented design for the preprocessing phase. The complete design of the preprocessing phase is not given for the sake of brevity. The overall design of the preprocessing phase is explained in the following paragraphs.

The important classes are described as follows:

- PreProTre – holds all preprocessing algorithms except the initial node numbering applied to the structure.
- EleSet – stores the element and node connectivity information for the structure.
- EleSubSet – stores a subset of elements in the EleSet, includes all elements in the model if there is no partitioning algorithm
- PreProAlgo – abstract base class for all preprocessing algorithms.
- PreProResult – any class derived from PreProAlgo stores its results in this class. It provides a common interface for storing the results of preprocessing algorithms.
- EliTree – stores the elimination tree constructed by the preprocessing phase

Figure 3.11 shows the interaction between the main classes designed to allow a flexible preprocessing phase. All preprocessing algorithms are derived from a single parent class called PreProAlgo. There are mainly two types of preprocessing algorithms: partitioning algorithms and matrix ordering algorithms. The partitioning algorithms subdivide the elements stored in the EleSubset class. The matrix ordering algorithms, on the other hand, condense a set of elements in the EleSubset class. The preprocessing algorithms that are applied to a structure are stored in a tree structure stored in PreProTree class. PreProTree holds the PreProTreeNode objects, which stores the EleSubset and the preprocessing algorithm that will be applied to an EleSubset. If a partitioning algorithm is assigned to a tree node, then the tree node also stores a bottom-up ordering algorithm to order separators found during the partitioning.

The results from the prepossessing algorithms are stored in the class called PreProResult. This class is responsible for updating the EleSubset according to the results from preprocessing algorithms. By the use of PreProResult, a preprocessing algorithm does not directly update the EleSet. This design simplifies implementation of new partitioning, coarsening, and matrix ordering algorithms since a preprocessing algorithm class is loosely coupled with the rest of the system.

An elimination tree [73] is built after the preprocessing is completed for the structure. An EliTree class instance keeps the elimination tree for the pivot-ordering found in the preprocessing phase. Namely, EliTree object stores the final result of the preprocessing phase, and other components of the solver package accesses EliTree object. The analysis and numerical solution phases do not access any other objects described in this section.

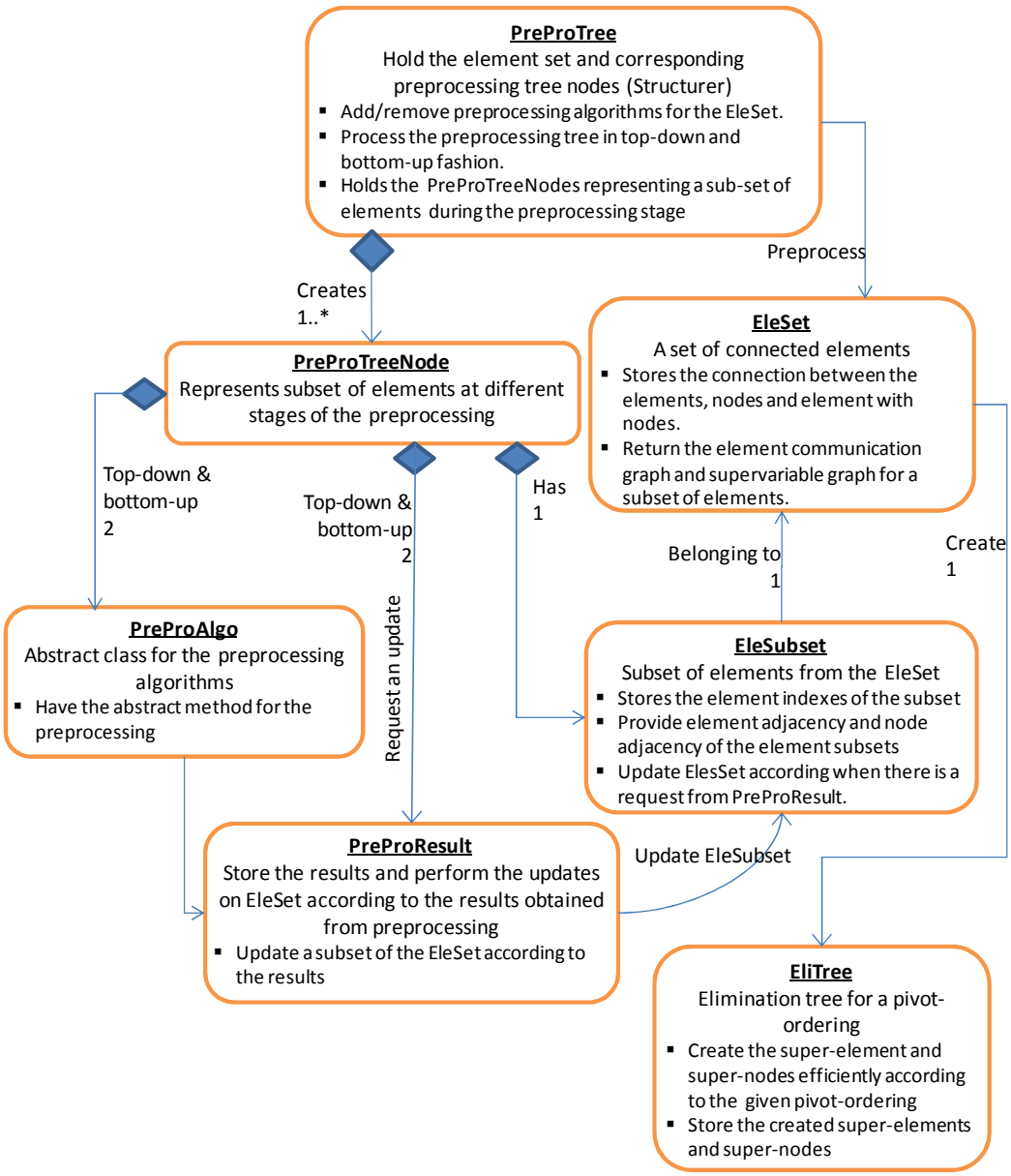


Figure 3.11: Main classes for the preprocessing package of the SES solver.

CHAPTER 4

ANALYSIS PHASE

After a pivot ordering is found in the preprocessing phase, a solution strategy and the data structures for the numerical solution are constructed in the analysis phase. The analysis phase constructs the assembly tree, builds the data structures for the numerical solution, determines the memory requirements, estimates the time required for the numerical factorization, and assigns an appropriate amount of work to each thread for the numerical solution.

4.1 Data Structures

4.1.1 Assembly Tree

The assembly tree represents the dependency between the partial factorization tasks [73]. The assembly tree is similar to the elimination tree except that the assembly tree typically contains fewer nodes since some nodes in the elimination tree are merged while building the assembly tree. The tree library developed by Peeters [137] is used for the implementation of the assembly tree. Figure 4.1 shows the tree implementation designed by Peeters [137]. The tree implementation is based on the linked lists. Here, doubly linked lists are used to store previous and next siblings of the tree nodes. In addition, a parent node holds two pointers to its first and last child. The assembly tree data structure allows building and modifying the assembly tree easily. Several tree traversal algorithms are also implemented in the tree library. Various tree traversals can be performed by using tree iterators that are similar to the iterators in standard c++ library.

In the SES solver package, assembly tree nodes store information required for the mapping, factorization and triangular solution algorithms. The information stored in the assembly tree nodes is given as follows:

- Eliminated and remaining nodes – The partial factorization on a frontal matrix is the condensation of the dofs corresponding to the eliminated nodes stored at the assembly tree node. The remaining nodes have the remaining dofs after the condensation. The number of eliminated and remaining nodes can be found efficiently by using the algorithm described by Gilbert et al. [138]. For a coefficient matrix with *nz* nonzero entries, the time complexity of their algorithm is almost linear. The eliminated and remaining nodes are found in the preprocessing phase during the construction of the elimination tree.
- Estimated partial factorization time – The partial factorization times are found by using a performance model. The time estimations are used in the mapping algorithm.
- Estimated subtree partial factorization time – The subtree partial factorization times are found by summing up the partial factorization times of the children assembly tree nodes. The time estimations are used in the mapping algorithm.
- Thread ID – The thread ID that will perform the numerical and symbolic factorization and triangular solution is stored in the assembly tree node.
- Number of threads used in the partial factorization and triangular solution of a frontal matrix – Multithreaded MKL kernels are employed for the partial factorization of some of the assembly tree nodes. The number of threads for the multithreaded MKL is stored in the tree nodes.
- FE's associated with the partial factorization of the frontal matrix – The FE's associated with a tree node are found in the preprocessing phase while constructing the elimination tree for a given pivot-ordering. Prior to the partial

factorization corresponding to a tree node, the FE's are assembled into the frontal matrix.

- The local indices for the parent frontal matrix – The local indices are used for the assembly of the children update matrices and FE's associated with the frontal matrix. The local indices are further explained in Chapter 4.1.4.
- Synchronization requirements (optional) – If a thread should wait for other threads before the partial factorization of a tree node, then this information is stored in the tree node.

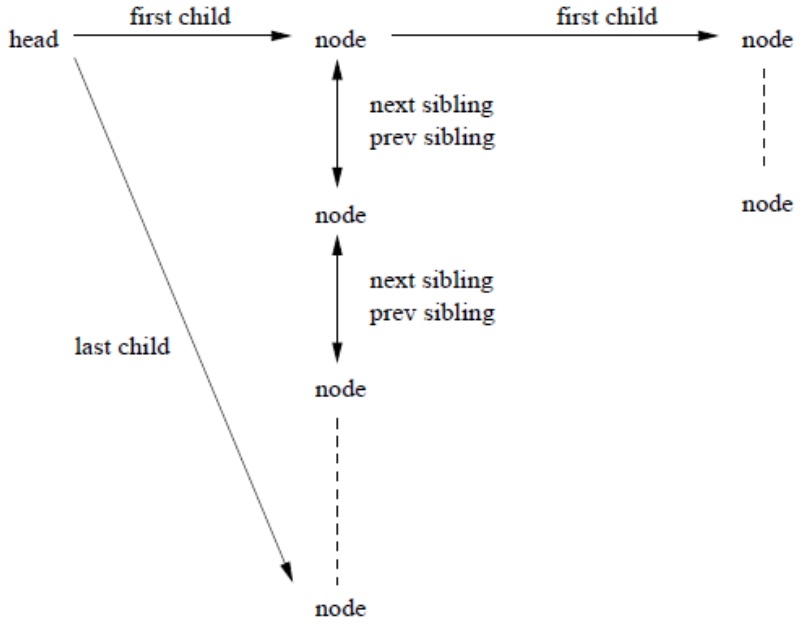


Figure 4.1: The implementation of the assembly tree. from the tree.hh library documentation [137].

Figure 4.2 illustrates the assembly tree using a 4×4 mesh. Here, FE's are shown in gray colour and they are not considered as assembly tree nodes. The FE's are stored at the leaves of the assembly tree. The black nodes shown in Figure 4.2 are assembly tree nodes for a nested dissection matrix ordering. The information stored in two example assembly tree nodes is also shown in Figure 4.2. Here, Node-24 is a subtree node. The

independent subtrees are assigned to threads for parallel factorization. A node within a subtree assigned to a thread is referred to as subtree node. Synchronization between threads is not required prior to the partial factorization of the subtree nodes. On the other hand, Node-25 is a high-level tree node. A tree node is referred to as high-level tree node if it is above the independent subtrees assigned to the threads. All threads should complete their partial factorization steps before the partial factorization starts at the Node-25. Therefore, Node-25 stores the information that synchronization between the threads is required prior to its partial factorization. As described in the subsequent Chapter, the numerical factorization, forward elimination, and back substitution algorithms each require a single synchronization point. Therefore, there is a single tree node that requires synchronization prior to the partial factorization of the corresponding frontal matrix.

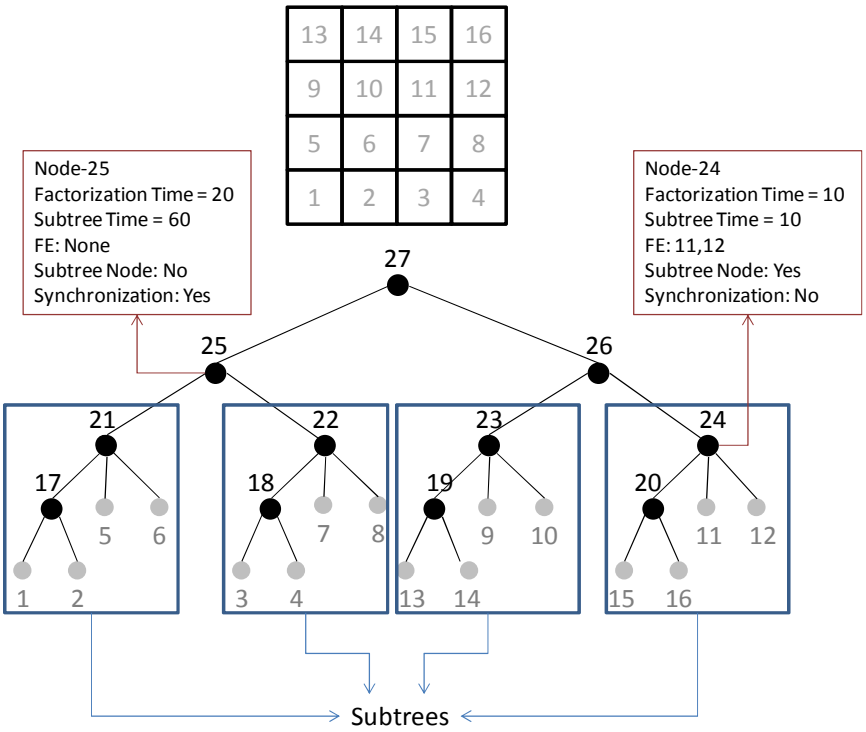


Figure 4.2: Assembly tree structure for the example 4×4 mesh. The gray nodes represent the finite elements in the model. There are four subtrees processed by different threads.

The assembly tree is built using the elimination tree constructed in the preprocessing phase. The nodes of the elimination tree are merged if the merge does not introduce any logically zero entries in the factors. This is referred to as the fundamental supernode partitioning by Ashcraft [57]. It is possible and often desirable to relax the condition of no logical nonzero and allow additional logically zero entries in the factors. This is called node amalgamation and is further discussed in Section 4.2.

4.1.2 Supervariables

The supervariables are a set of variables (dofs) that are connected to the same element set. Compared to the use of original variables, the use of supervariables reduces the analysis phase execution time. Figure 4.3 shows the supervariables for a subset of elements from a FE mesh. Here, four nodes are shared between Element-1 and Element-2. Instead of treating the dofs for four nodes separately, we can treat them as a single entity, which is called a supervariable. The supervariables are found for an input structure. It is also possible to perform a search to find the new supervariables formed after some elimination steps. However, Reid and Scott [139] stated that finding supervariables after each elimination step is a costly operation and increases the analysis phase execution time.

In finite element models, dofs corresponding to each node belongs to the same supervariable. The analyze phase of the SES solver package exploits supervariables by working with the nodes instead of the dofs. The assembly tree construction, node amalgamation, and local index computations (Chapter 4.1.4) are performed using the nodes instead of dofs. In most FE analyses, the number of dofs is the same at all the nodes. For example, there are 6 dofs at each node except from the nodes having restraints for 3D frame analysis. This structure is exploited in the SES solver package by storing the number dofs for the nodes that are partially restrained in an array. For the remaining nodes with the same number of dofs, only a single variable holds the number of dofs.

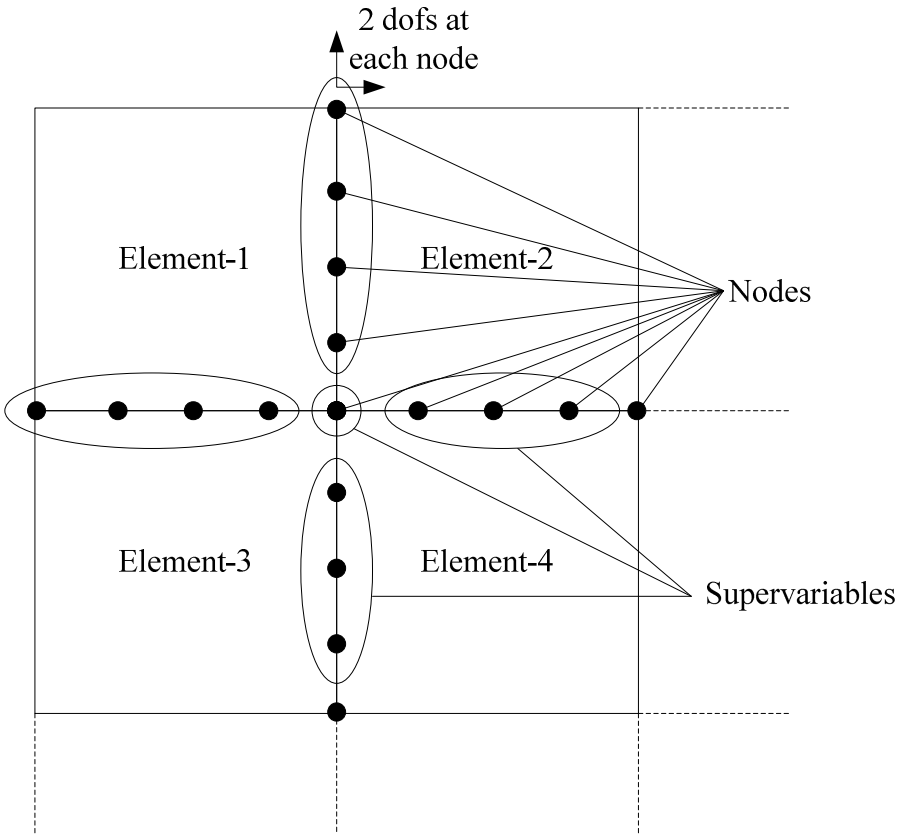


Figure 4.3: The supervariables for four elements isolated from the rest of a FE mesh.

4.1.3 Factors, Frontal Matrix, and Update Matrix Stack

A multifrontal method is employed for the numerical factorization. In the multifrontal method, partial factorizations are performed on dense frontal matrices. Once the factors corresponding to a pivot column block are computed, they are not accessed again until the triangular solution. Therefore, entire factors are not required to be stored in main memory. This section discusses the frontal matrix and the data structures related to the multifrontal method. The numerical factorization and triangular solution uses the data structures. However, the memory requirements for the data structures are determined in the analysis phase.

The frontal matrix is mainly composed of three components: diagonal factors, $\mathbf{L}_B$, off-diagonal factors, $\mathbf{L}_{off}$, and Schur complement, **S** (see Chapter 5 for a detailed

description of each component). Once the partial factorization is completed for a frontal matrix, only **S** is required for the subsequent partial factorization steps. If the partial factorizations of the assembly tree nodes are performed in a postorder tree traversal order, then a stack data structure can be used to store **S**. As stated previously, after the partial factorization of a frontal matrix, diagonal and off-diagonal factors are not required until the triangular solution.

Figure 4.4 shows the data structures used for the multifrontal factorization. We design three classes for the abstraction of the three items shown in Figure 4.4. Here, the frontal matrix class physically stores the **S** matrix only. Our frontal matrix implementation does not hold the factors physically. The frontal matrix has a reference to the factors, which are stored in a separate memory location. Frontal matrix objects have a pointer to the memory location storing the corresponding factors. A distinct class is responsible for storing the factors and providing pointers for the frontal matrix (*Factors* in Figure 4.4). Once the partial factorization is completed, no data copy is required for the factors since the frontal matrix works with the pointers. On the other hand, **S** is copied to the update matrix stack. Although $\mathbf{L}_B$ and **S** are symmetric matrices, packed storage scheme is not employed for the partial factorization operations on these matrices. This is due to the performance consideration for MKL Cholesky decomposition [140] and the unavailability of a BLAS3 rank-k update subroutine for packed storage scheme. However, the partial factorization operations are performed for only lower diagonal entries. In addition, only **S**, the lower diagonal part of the Schur complement, is copied to the update matrix stack to save some memory for the storage of the update matrices.

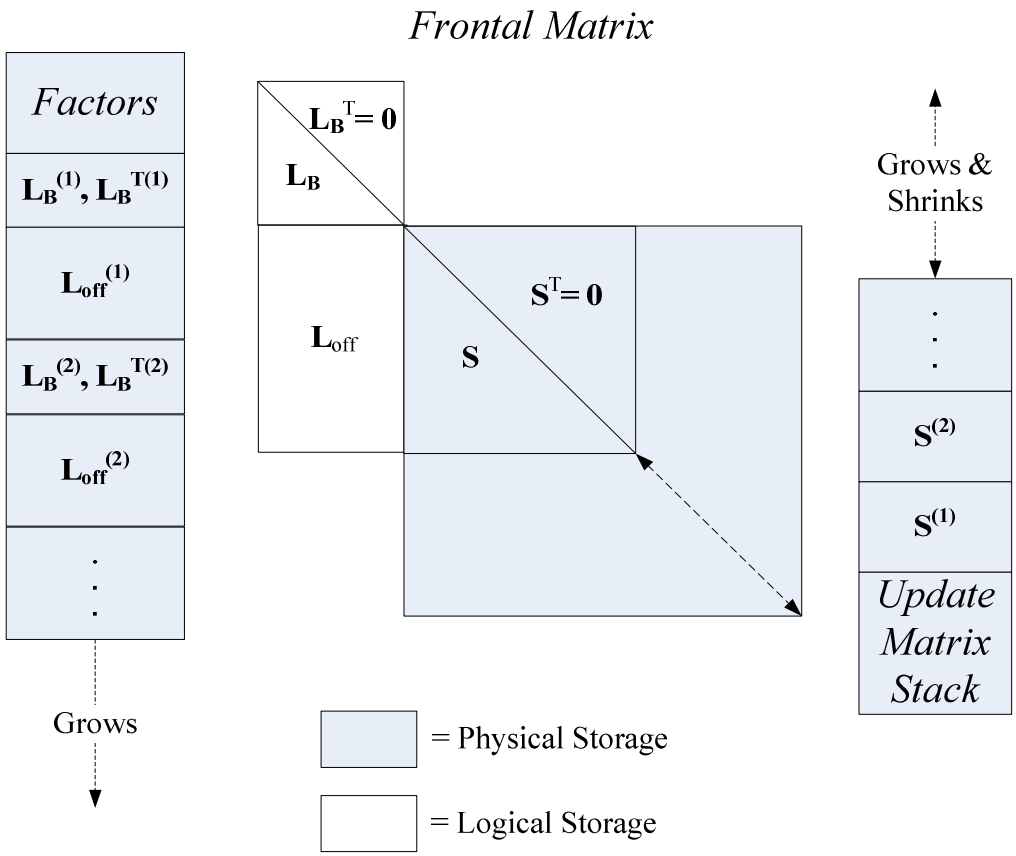


Figure 4.4: Data structures used for multifrontal method.

The size of the frontal matrix depends on the number of eliminated and remaining dofs for an assembly tree node and it varies during the numerical factorization. The size of the update matrix stack also varies during the numerical factorization depending on the number of active frontal matrices at a certain point during the factorization. At any point during the factorization, the summation of the frontal matrix size and update matrix stack size is the active memory requirement. Assuming that calculated factors are stored on disk, the maximum of such active memory requirements is the total memory requirement of the multifrontal method. This assumes that the memory is perfectly shared between the frontal matrix and update matrix stack during the factorization steps. It should be noted that the current implementation of the SES uses separate memory locations for the frontal matrix and the update matrix stack. Therefore, there is no overlap between the frontal

matrix memory and update matrix memory. Consequently, the active memory requirement for the SES solver package is the summation of maximum frontal matrix size and maximum update stack size.

4.1.4 Frontal Matrix Indices

Prior to the partial factorization of a frontal matrix, the update matrices corresponding to the children tree nodes are assembled into the frontal matrix. The assembly of the update matrices are performed using the eliminated and remaining dof indices stored in the parent and children assembly tree nodes. As described previously, the node indices are used instead of the dof indices in order to improve the speed and memory requirements of analysis phase. There are two types of indices associated with a frontal matrix: global and local indices. The global indices are the eliminated and remaining node numbers for the partial factorization (or condensation) of a frontal matrix. They are stored in two sorted arrays for eliminated and remaining nodes. Sorting the node indices allows merging the node indices of two tree nodes in linear time. The local indices are used for the assembly of the children update matrices. The local indices are the location of an update matrix entry at the frontal matrix of the parent assembly tree node. The local indices are found by using the sorted node indices. Finding the local indices can be performed in linear time for sorted global node indices at parent and children tree nodes. For an assembly tree node, the local indices are found for the remaining nodes only. Figure 4.5 shows the global and local indices for example assembly tree nodes. The eliminated nodes are marked with blue color and node numbers inside the matrix shows the global indices for the example assembly tree nodes. The local indices for children nodes are also shown at the left of the children frontal matrices.

The local indices are used for the assembly of the children update matrices if node blocking is not performed for the parent frontal matrix (see subsequent Section 4.3 for a detailed description of node blocking). The local indices are also found for each finite

element required for the partial factorization of a frontal matrix. Element stiffness matrices are assembled at the numerical factorization phase by using the local indices found in the analysis phase. For large frontal matrices, it is more efficient to perform the update matrix assembly operations on dense matrix blocks instead of individual matrix entries. For this purpose, nodes are partitioned into node blocks, which are the node sets that are adjacent at the frontal matrices corresponding to parent and the child assembly tree nodes. Node blocks can be assembled to the frontal matrix efficiently by using BLAS kernels. The node blocks are shown in Figure 4.5 for the local indices at the children tree nodes. There are two node blocks for each child, which are shown within the rectangular boxes in Figure 4.5. Section 4.3 further explains the node blocking and how the node blocks are found based on the local indices.

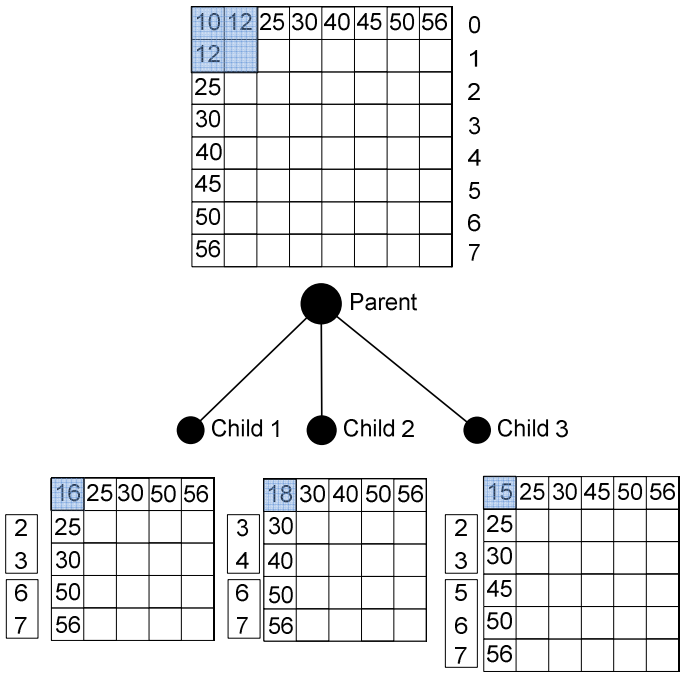


Figure 4.5: The global and local indices for the example assembly tree nodes. The local indices for children are shown on the left of the frontal matrices.

4.2 Node Amalgamation

The creation and storage of the update matrices is the overhead required for the use of dense matrix kernels in the multifrontal method. The overhead for creating small frontal matrices and storing their update matrices can be reduced by the amalgamation of tree nodes with a small number of eliminated and remaining dofs. Various amalgamation schemes are proposed in the literature. Ashcraft and Grimes [57] merged tree nodes if the number of logical zero entries introduced by the amalgamation is below a threshold value. New supernodes created by the node amalgamation are called relaxed supernodes. Duff [141] merged the nodes in the assembly tree if the number of eliminated variables at a tree node is less than a threshold value. Reid and Scott [139] merged a child node with its parent if number of eliminated variables at the parent and child nodes are both less than a threshold value. They reported that this criterion gave better results compared to checking the number of variables at the child node only. The logically zero entries introduced to the factors are not taken into account in their node amalgamation scheme.

In this study the node amalgamation criterion given by Reid and Scott [139] is employed. A child node is merged with its parent if the number of eliminated dofs is smaller than a certain value (*smin*) for both of them. The nodes of the assembly tree are visited in pre-order traversal fashion and the amalgamation criteria are checked for all children of a visited node. If the nodes are visited in a pre-order tree traversal for the node amalgamation algorithm, then this usually gives fewer update operations compared to the postorder tree traversal. In our amalgamation scheme, a pre-order tree traversal is used and the children of a tree node are visited in an arbitrary fashion. The amalgamation criterion is rechecked for an assembly tree node that is amalgamated. Further amalgamation is performed for an amalgamated tree node if it can be merged with its new children.

If *smin* is too large, then the number of extra arithmetic operations required for logically zero entries will overshadow the benefits of node amalgamation. There is no

single *smin* value that gives the best results for all types of problems. Numerical experiments are required to determine an optimal range for the parameter. Duff [141] reported modest gains in factorization times for different amalgamation parameters. Moreover, the critical value for node amalgamation was not obvious in his study. Duff [141] concluded that the amalgamation is not essential and good performance can be obtained without amalgamation.

We illustrate the impact of node amalgamation using the problem f500×500. Table 4.1 shows the effect of node amalgamation for the example problem. As shown in Table 4.1, the number of update operations decreases constantly as we increase *smin*. The node amalgamation decreases the factorization times up to *smin*=25. As *smin* gets larger than 25, the factorization times tend to increase. In Chapter 6, further numerical experiments are performed in order to determine the optimal amount of node amalgamation.

smin	**Number of Tree Nodes**	**Flop (10^9) for Factorization**	**Update Size (10^8)**	**Factorization Times (sec)**
0	192829	46.05	1.92	10.71
5	147955	46.19	1.77	10.42
10	95468	46.52	1.56	9.85
15	82970	46.68	1.51	9.83
20	70755	47.09	1.45	9.79
25	59237	47.61	1.38	**9.65**
30	55326	47.91	1.35	9.67
35	47707	48.52	1.3	9.9
40	42741	49.17	1.25	9.88

Table 4.1: Effect of node amalgamation for f500×500

It should be noted that the SCOTCH matrix ordering subroutines allow inputting the minimum size of the supernodes (parameter *cmin* determines the minimum size of the supernodes for each pivot). In this case, node amalgamation typically is not necessary since the number of eliminated nodes at the tree nodes is guaranteed to be larger than *cmin*. However, this option for SCOTCH matrix ordering subroutines is not used since it typically yields larger factorization times compared to the explicit node amalgamation.

4.3 Node Blocking

The initial version of the multifrontal code assembled the update matrices into the frontal matrices by using the local indices (see Section 4.1.4). In this case, for the assembly of each update matrix entry, the row and column indices of the entry is first read from the array storing the local indices. The code that uses the local indices for the assembly of update matrices is shown in Figure 4.6. The indices for the entries in the update matrix are read at lines 9 and 10 in Figure 4.6. This is indirect addressing and it is not efficient for cache hierarchy. In addition, the above code has three branches (if statements at lines 11, 18 and 19), which is not desirable for machines with deep instruction pipeline architectures. Finally, the access to the frontal matrix entries at line 20 is performed in a random fashion, which prevents exploiting the instruction level parallelism or SIMD instruction sets.

The profile information of the multifrontal code is shown in Table 4.2 for the test problem f500×500. As shown in this table, most of the time (64.6%) is spent on the MKL BLAS/LAPACK kernels. The subroutine that takes the most of the time after the MKL functions is the update matrix assembly code shown in Figure 4.6. This subroutine takes 18.9% of the total factorization time. Finally, a significant time is spent in memory copy and memory set operations (memcopy and memset). The memory operations are mainly used to copy Schur complements to the update matrix stack.

```
1 AssembleToLowerDiagonal(...)
2 {
3  ef::Int_t iTo;
4  ef::Int_t jTo;
5  for(ef::UsInt_t i = 0 ; i<assemblyIndicesNum ; ++i)
6  {
7   for(ef::UsInt_t j = i ; j<assemblyIndicesNum ; ++j)
8   {
9    iTo = assemblyIndices[i];
10   jTo = assemblyIndices[j];
11   if( jTo < iTo ){
12    std::swap(iTo,jTo);
13   }
14
15    iTo -= columnOffset;
16    jTo -= rowOffset;
17
18    if( iTo<m_ColNum && jTo<m_RowNum ){
19     if( iTo>=0 && jTo>=0 ){
20          m_Entry[ iTo*m_LeadingDim + jTo ] +=
21                         *packedMatrixEntries;
22     }
23    }
24    ++packedMatrixEntries;
25   }
26  }
27}
```

Figure 4.6: The c++ code that uses local indices for the assembly of update matrices.

Subroutine(s)	% of the Total Factorization Time
BLAS/LAPACK	64.6
AssembleToLowerDiagonal	18.9
memset	5.4
memcopy	4.2
Other	6.9

Table 4.2: Profile information for the multifrontal factorization of f500x500, without node blocking

The linear algebra subroutines in MKL are already optimized and further optimization of memory copy and set operations are out of the scope of this study. However, the time spent in the AssembleToLowerDiagonal function given in Table 4.2 can be reduced. For example, instead of reading the index of each element, the elements of the update matrix can be assembled in continuous dense matrix blocks. If the blocks are used for the assembly, the index of a block is read once and all entries within the block are assembled to the corresponding location in the frontal matrix. The larger the block sizes are the more efficient the assembly operations will be. This will also allow the use of BLAS1 subroutines for the assembly of the update matrices. However, in order to fully exploit the blocked assembly operations a node blocking algorithm is required. The node blocking algorithm reorders the nodes in frontal matrices so that the assembly operations can be performed on continuous data blocks. An algorithm that reorders and groups the local indices of a frontal matrix is designed and implemented to allow performing the update matrix assembly operations on large continuous data blocks.

The blocking algorithm reorders the nodes in parent and children tree nodes so that the local indices are continuous at the children assembly tree nodes. First, the nodes at a parent assembly tree node are partitioned into disjoint sets according to the children tree nodes that contain the nodes. Each set represents a unique combination of the children tree nodes that contains all nodes in the set. Therefore, the total number of sets is equal to 2^N, where N is the number of children tree nodes. The nodes within each set correspond to a node block for which assembly operations can be performed all at once. The sets are ordered to maximize the adjacent node blocks at the children tree nodes. The blocking can be performed recursively for the children of the assembly tree node. However, the node blocking for the child tree node must consider the ordering of the node sets found at the parent tree node.

The node blocking algorithm is illustrated with a simple example shown in Figure 4.7. Figure 4.7 shows the root of an example assembly tree and its three children nodes.

The local indices for the children tree nodes are also shown in Figure 4.7. There are 8 node sets since the example assembly tree node has three children (2^3=8). The nodes within the node sets are shown in Figure 4.8. After node sets are found, the sets are ordered in the increasing lexicographic order according to the set names given at the bottom of the Figure 4.8. The name of a set is given according to the index of the children tree nodes that contains the nodes in the set. For example, if the nodes only belong to child 1, then the name of the set is Set-001. Similarly, if the nodes belong to child 1 and child 2 only, then the name of the set is Set-011. The lexicographic ordering usually yields continuous node blocks for the tree nodes with small number of children. The node sets that contain nodes from a child tree node are assigned to that tree node according to the lexicographic ordering of the node sets. The node blocking is performed recursively for the children tree nodes. The sets are found in the same fashion for the eliminated nodes at the children. However, for the remaining nodes at a children assembly tree node, a hierarchical structure of the node sets are constructed in order to preserve the ordering found at the assembly tree node. For the remaining nodes at a children tree node, the node sets are found for each node set found at the parent assembly tree node. A tree structure is used to store the hierarchical node set information. The head of each node set tree represents the first creation of a node set, which corresponds to a subset of eliminated nodes at an assembly tree node. The children of a node at the node set tree are the node sets found at the children assembly tree nodes.

Once all node sets are found and are lexicographically ordered, the nodes in the frontal matrices are renumbered starting from the leaf nodes of the node set trees. Figure 4.9 shows the node indices in the frontal matrices of the parent node and children nodes after the node blocking is applied for the illustrative example shown in Figure 4.7. In Figure 4.9, the parent frontal matrix entries that receive update from different children tree nodes are marked with different colours. As shown in Figure 4.9, the assembly of child 3 benefits most from the node blocking. After applying the node blocking, the entire

update matrix for child 3 can be assembled to the frontal matrix by making a single call to BLAS3 dense matrix addition subroutine. Although it looks like that children 1 and 2 do not benefit much from the node blocking, this is due to the small number of nodes used for the sake of this simple illustrative example. For sufficiently large frontal matrices, the blocking improves the efficiency of update operations for all children nodes.

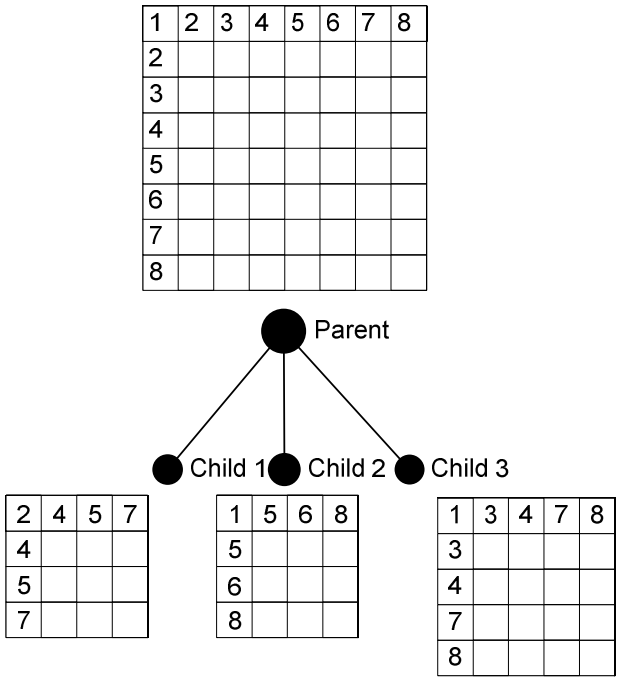
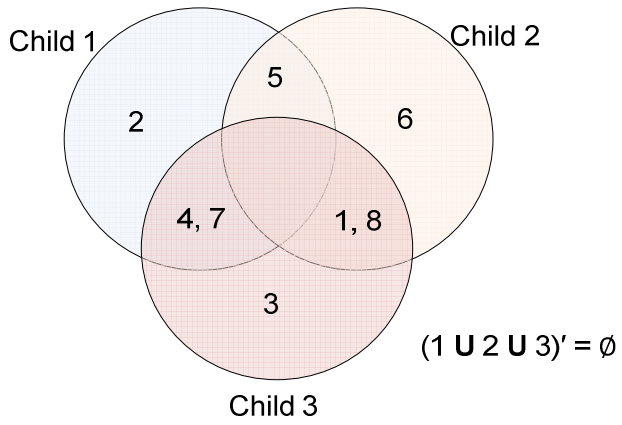


Figure 4.7: An example assembly tree node and its children. The parent node is the root of the assembly tree. The local indices for the remaining nodes at the children are shown.



Set-001 = {2} Set-010 = {6} Set-011 = {5}

Set-100 = {3} Set-101 = {4,7} Set-110 = {1,8}

Figure 4.8: Node sets for the remaining nodes at the children for the example tree nodes.

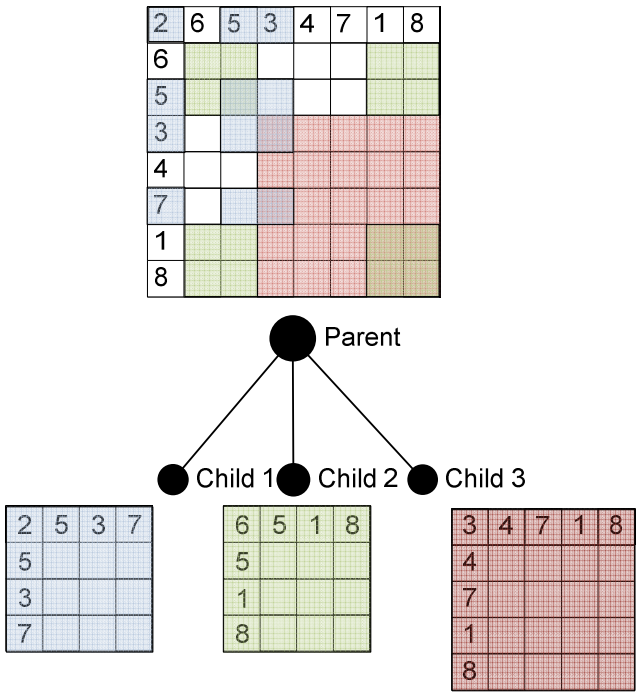


Figure 4.9: Node blocks found for the example assembly tree nodes

Table 4.3 shows the profile information of the multifrontal factorization after implementing the node blocking. If the code profile information shown in Table 4.3 is compared with the one in Table 4.2, we observe that the time spent in the assembly of the update matrices is reduced from 18.9% to 3.9% after employing the node blocks for the assembly of the update matrices. In addition, the blocked assembly increases the time spent in the optimized BLAS library as shown in Table 4.3.

Subroutine(s)	**% of the Total Factorization Time**
BLAS/LAPACK	79.5
BlockedAssembly	3.5
Memset	4.6
Memcopy	4.4
Other	8

Table 4.3: Profile information for the multifrontal factorization of f500×500 with node blocking

The time required for node blocking may be large for the tree nodes with a large number of children nodes since the number of node partitions grows exponentially with the number of the children nodes. Fortunately, the assembly trees for the FE problems generally have a small number of children. Therefore, the exponential growth is usually not a problem. However, the node amalgamation may flatten the assembly tree, creating parent nodes with a large number of children nodes with small frontal matrices. Considering that the tree nodes with large number of children nodes typically have small frontal matrices, the node blocking is not performed for the frontal matrices smaller than a threshold value. A cut off value for the node blocking, *blkmin*, is used as the stopping criteria for the node blocking. The node blocking is not applied to a tree node and all of its descendants if the size of the frontal matrix is smaller than *blkmin*. Table 4.4 shows factorization and analysis times for the problem f500×500 with alternative *blkmin* values. The node amalgamation parameter *smin* is taken as 10 for the example problem. As

shown in Table 4.4, as the *blkmin* value increase the analysis time decreases since fewer node blocks are found for smaller *blkmin* values. The bottom row of Table 4.4 shows analysis and factorization times with no node blocking. As shown in Table 4.4, the factorization time tends to increase even for small *blkmin* values.

According to execution times given in Table 4.4, a *blkmin* value can be chosen to minimize analysis plus factorization phase times. For the current implementation of the solver package, numerical experiments show that a cut-off value of 50 usually gives satisfactory analysis plus factorization times and yet the factorization performance is not compromised significantly (see Chapter 6.6 for the details). However, the current implementation of the analysis phase is not optimized. The analysis phase execution time may be insignificant compared to the overall execution time of the solver for an optimized analysis phase.

blkmin	**Analysis Time (sec)**	**Factorization Time (sec)**
0	1.89	9.28
25	1.83	9.30
50	1.43	9.37
75	1.19	9.49
100	1.07	9.5
125	1.03	9.51
150	1.00	9.59
∞	0.96	10.16

Table 4.4: The cut off point for the node blocking for the test problem f500×500

4.4 Estimation of the Factorization Time

As it is discussed in the subsequent Chapter 4.5, partial factorization time estimations of the assembly tree nodes are used to find a subtree to thread mapping that minimizes the factorization time. Moreover, factorization time estimations are helpful to monitor the performance of the multifrontal solver. The actual factorization times can be compared with the estimated factorization times to determine any unanticipated

performance degradations. For example, the performance can be hindered by computational resource limitations, such as cache conflicts. Finally, predicting the factorization time prior to the actual factorization is a user-friendly feature for a direct solver. The user may want to build a simpler finite element model to get the results in a timely fashion if the predicted factorization time is too large. Alternatively, in order to reduce the factorization time, an alternative preprocessing strategy may be employed, or a system with higher computational power can be used for the FE analysis.

4.4.1 Partial Factorization Time

We use the experimental speed of BLAS/LAPACK subroutines used for the partial factorization in order to estimate the partial factorization time for a frontal matrix. The partial factorization speeds are determined by executing BLAS/LAPACK subroutines on the test system. The BLAS/LAPACK subroutines are executed for hypothetical frontal matrices with different numbers of eliminated and remaining variables. The partial factorization times are recorded for the execution of the corresponding MKL subroutines. The partial factorization times are too small to measure with sufficient accuracy for the partial factorization of a small frontal matrix. In order to measure the execution time accurately, partial factorization is performed for multiple times within a loop. A large number of repetitions may be required to measure the partial factorization times of the small frontal matrices with sufficient accuracy. Then, the average execution time is found by dividing the total execution time of repeated partial factorizations to the number of repetitions. The partial factorization speeds of frontal matrices that have less than 1000 remaining variables are shown in Figure 4.10. As shown in Figure 4.10, the speed of partial factorization varies greatly for small frontal matrices. For example, the speed varies between 1 GFlop/sec and 7 GFlop/sec for frontal matrices with 50 eliminated variables. Figure 4.11 shows the speed of partial factorization for frontal matrices with more than 1000 remaining variables. As shown in

Figure 4.11, the variation in the partial factorization speed is small for frontal matrices with large number of remaining variables. Figure 4.10 and Figure 4.11 also show that the partial factorization performance increases as the frontal matrix size increase. This is true even for frontal matrices with 50 eliminated variables. The theoretical machine peak speed is 9.6 GFlop/sec for a single core of the test system. As shown in Figure 4.11, the partial factorization runs at 8.6 GFlop/sec for sufficiently large matrices, which is close to the machine peak speed.

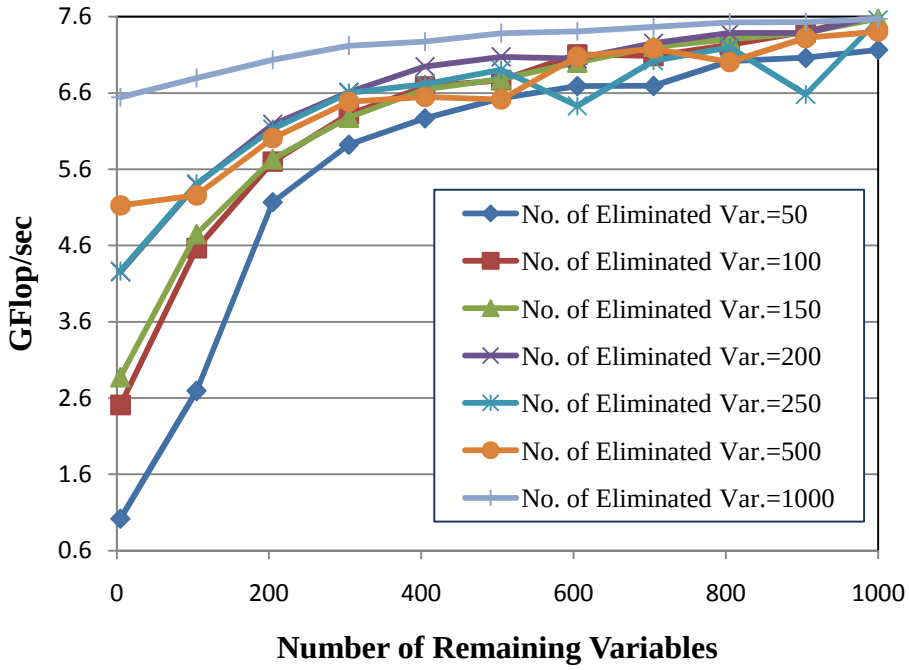


Figure 4.10: Performance of partial factorization, up to 1000 remaining variables.

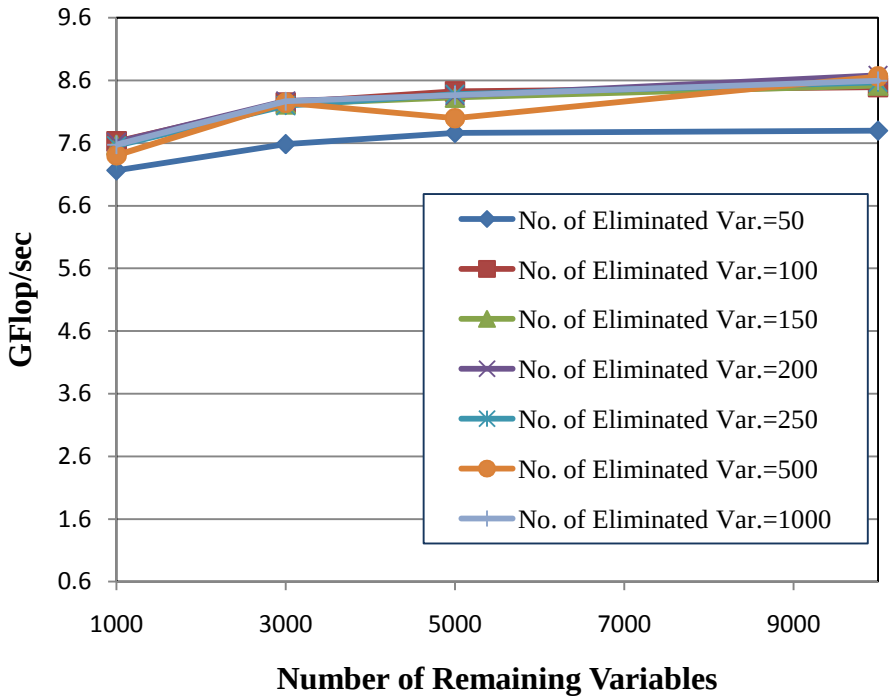


Figure 4.11: Performance of partial factorization, between 1000 and 10000 remaining variables.

The partial factorization time for a frontal matrix is found by dividing the theoretical operation count for the partial factorization by the approximated speed for the partial factorization. The partial factorization speed of a frontal matrix is estimated based on the known partial factorization speeds of the frontal matrices with similar number of eliminated and remaining variables. The experimental partial factorization speeds are stored in a table. Table 4.5 shows an example table constructed based on a small set of experiments on the test system. Each entry in this table is the speed of partial factorization in terms of GFlop/sec for a frontal matrix with a specific number of eliminated and remaining variables. The columns and rows of Table 4.5 correspond to number of eliminated and remaining variables respectively. In order to determine the speed of a frontal matrix, first, the speed of frontal matrices with similar number of eliminated and remaining variables are read from the table. Next, a piece-wise linear

approximation is performed to estimate the partial factorization speed. Figure 4.12 illustrates the piece-wise approximation of the partial factorization speed with x' eliminated variables and y' remaining variables. In Figure 4.12, x-axis is the number of eliminated variables and the y-axis is the number of remaining variables. The z values are the partial factorization speed. Known partial factorization speeds are shown with dots. We approximate z' (the estimated speed for a given frontal matrix) based on the known values of partial factorization speeds. The frontal matrices with the closest number of eliminated and remaining variables are used for the approximation. In Figure 4.12, the closest numbers of eliminated and remaining variables are: (x_1, y_1), (x_2, y_2), (x_3, y_3), and (x_4, y_4). The Lagrange polynomials given in Equation 4.1 are used to determine the approximate speed z'.

The partial factorization times are recorded for a number of frontal matrices to determine the known values of the partial factorization speeds. As shown in Figure 4.10, the performance of MKL functions varies greatly for small numbers of eliminated and remaining variables. Therefore, a large number of test runs with small increments of number of eliminated and remaining variables are performed to capture the behaviour of the partial factorization speed within the range of [2,500] eliminated variables.

Number of Remaining Variables	Number of Eliminated Variables			
	2000	3000	4000	5000
0	7.69068	8.0135	8.14555	8.25766
1000	8.07826	8.23716	8.29783	8.33763
2000	8.29528	8.33914	8.34696	8.45212
3000	8.36739	8.34284	8.46438	8.49085
4000	8.44095	8.49388	8.50022	8.51074
5000	8.4862	8.52627	8.53178	8.5578

Table 4.5: Example table for partial factorization speeds of different frontal matrix sizes. Table values are given in GFlop/sec.

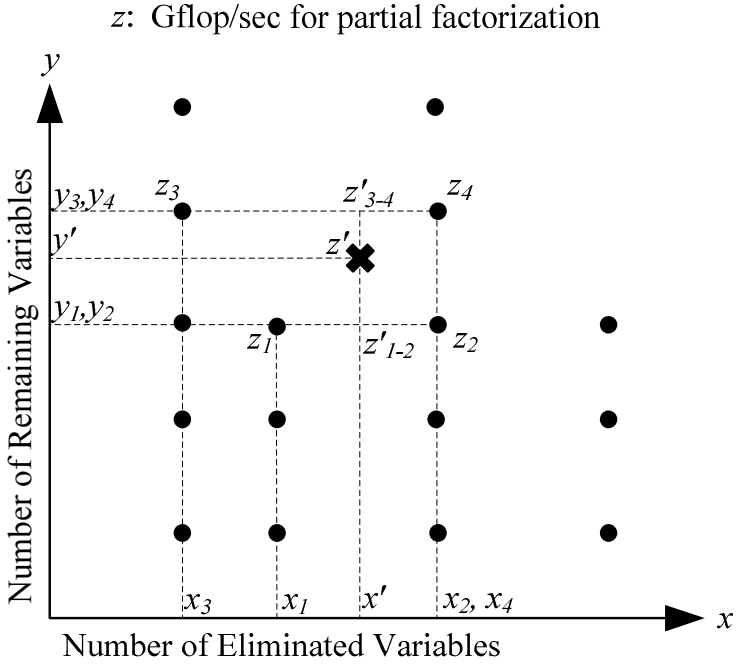


Figure 4.12: The approximation of the partial factorization speed, z', based on the known values of z.

$$z'_{1-2} = \frac{(x' - x_1)z_1 + (x_2 - x')z_2}{x_2 - x_1}$$

$$z'_{3-4} = \frac{(x' - x_3)z_3 + (x_4 - x')z_4}{x_4 - x_3} \qquad (4.1)$$

$$z' = \frac{(y' - y_1)z'_{1-2} + (y_3 - y')z'_{3-4}}{y_3 - y_1}$$

Table 4.6 compares the estimated and actual partial factorization times for different frontal matrix sizes. The frontal matrix sizes shown in this table are not used in the numerical experiments to determine the speed of the partial factorization, except for the rows with 500 eliminated variable. As shown in this table, on average, the partial factorization times is estimated with 0.56% of error. For the factorization of a FE mesh,

the execution time for the large matrices has a greater impact on the total execution time. Therefore, a small error is desirable for the large frontal matrices. As shown in Table 4.6, the error is small for both large and small frontal matrices.

Number of Eliminated Variables	Number of Remaining Variables	Actual Partial Factorization Time (milli-sec)	Estimated Partial Factorization Time (milli-sec)	% Difference
50	50	9.36E-02	9.55E-02	2.01
50	150	3.24E-01	3.24E-01	0.01
50	250	6.86E-01	6.74E-01	-1.72
50	350	1.16E+00	1.16E+00	0.02
50	450	1.77E+00	1.76E+00	-0.72
300	0	1.59E+00	1.60E+00	0.44
300	500	1.75E+01	1.74E+01	-0.61
300	1000	5.14E+01	5.09E+01	-0.91
300	1500	1.03E+02	1.02E+02	-0.77
300	2000	1.70E+02	1.71E+02	0.53
500	0	6.83E+00	6.58E+00	-3.77
500	500	3.92E+01	3.88E+01	-1.08
500	1000	1.02E+02	1.01E+02	-0.74
500	1500	1.93E+02	1.93E+02	0.38
500	2000	3.14E+02	3.13E+02	-0.25
650	0	1.49E+01	1.40E+01	-5.87
650	500	6.32E+01	6.14E+01	-2.87
650	1000	1.51E+02	1.49E+02	-1.22
650	1500	2.75E+02	2.73E+02	-0.77
650	2000	4.38E+02	4.36E+02	-0.46
1500	0	1.52E+02	1.52E+02	0.15
1500	2000	1.42E+03	1.41E+03	-0.26
1500	4000	4.03E+03	4.04E+03	0.41
1500	6000	7.99E+03	8.10E+03	1.41
2500	0	6.58E+02	6.64E+02	0.87
2500	2000	3.34E+03	3.33E+03	-0.11
2500	4000	8.32E+03	8.29E+03	-0.33
2500	6000	1.55E+04	1.56E+04	0.60
				Avg.: -0.56 Max: 2.01 Min: -5.87

Table 4.6: Partial factorization time estimations for executing MKL kernels with a single thread.

For the parallel factorization, the tree nodes close to the root of the assembly tree are processed using the multithreaded BLAS/LAPACK subroutines. The tree nodes close to the root node usually have a large number of eliminated variables compared to the lower levels of the assembly tree. Another set of numerical experiments is performed to determine the speedup of the MKL functions using four threads. The experiments are performed for frontal matrices having 50 or more eliminated variables. Figure 4.13 and Figure 4.14 show the experimental speedup for the partial factorization as a function of total flop required for the partial factorization. Figure 4.13 is for flop values smaller than 1 GFlop, and Figure 4.14 is for flop values larger than 1 GFlop. In addition to the experimental speedup, both figures show an approximation to the speedup. The approximation is found by fitting a power function to the experimental speedup values. The approximation for the speedup is given as follows:

$$speedup = 0.18 \cdot flop^{0.15} - 0.1$$

If $speedup < 0.9$ **then** $speedup = 0.9$

If $speedup > 3.7$ **then** $speedup = 3.7$

Four thread partial factorization times are estimated by multiplying the speedup approximation with the speed estimation for one thread partial factorization of a frontal matrix. Table 4.7 shows the accuracy of the predictions for partial factorization with four threads. The average error in estimations is 0.83% for the frontal matrix sizes shown in Table 4.7. As shown in Table 4.7, the partial factorization time is underestimated for large frontal matrices with a small number of remaining variables. It is important to accurately predict the partial factorization times of large frontal matrices since the partial factorization for large frontal matrices correspond to a significant portion of the total factorization time. In addition, the partial factorization is performed using multithreaded

MKL kernels for the high-level tree nodes. In order to improve the predictions, the speedup approximation can be modified by considering the number of remaining variables. However, in this study, the simple speedup approximation that is only a function of flop for partial factorization is used.

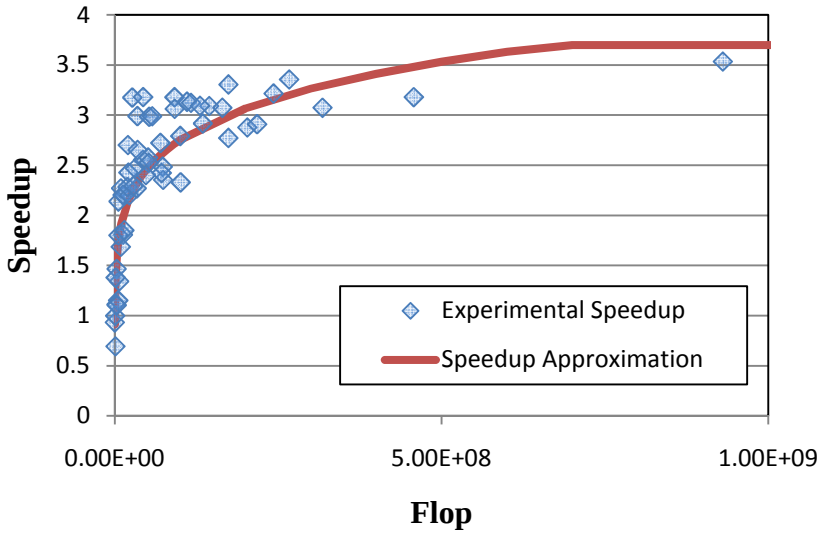


Figure 4.13: Partial factorization speedups using four threads (flop is between 0 and 1E9)

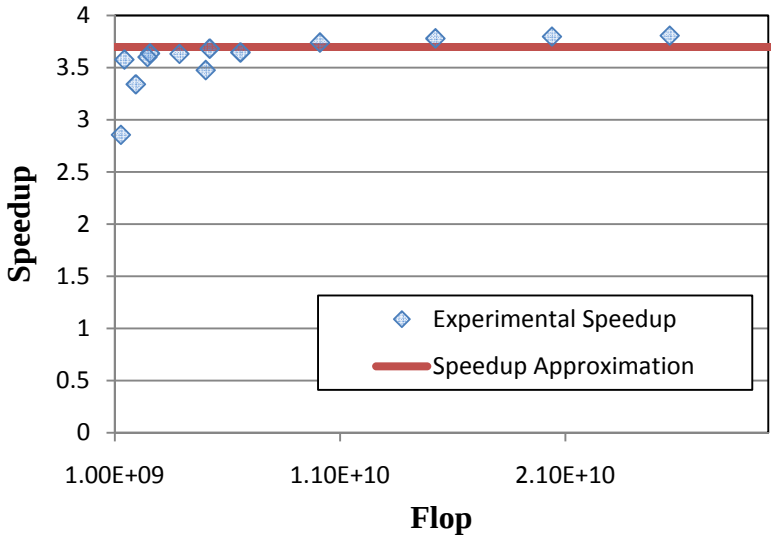


Figure 4.14: Partial factorization speedups using four threads (flop is between 1E9 and 3E9)

Number of Eliminated Variables	Number of Remaining Variables	Actual Partial Factorization Time (milli-sec)	Estimated Partial Factorization Time (milli-sec)	% Difference
50	50	9.90E-02	8.75E-02	-11.59
50	150	1.98E-01	2.27E-01	14.69
50	250	3.85E-01	4.09E-01	6.32
50	350	4.75E-01	6.36E-01	33.97
50	450	6.10E-01	9.00E-01	47.51
300	0	7.94E-01	8.46E-01	6.52
300	500	6.68E+00	6.08E+00	-8.93
300	1000	1.54E+01	1.49E+01	-2.86
300	1500	2.94E+01	2.75E+01	-6.37
300	2000	4.73E+01	4.62E+01	-2.27
500	0	2.82E+00	2.74E+00	-2.95
500	500	1.31E+01	1.19E+01	-8.79
500	1000	2.97E+01	2.74E+01	-7.81
500	1500	5.56E+01	5.23E+01	-6.04
500	2000	8.80E+01	8.46E+01	-3.82
650	0	5.65E+00	5.15E+00	-8.82
650	500	2.04E+01	1.76E+01	-13.95
650	1000	4.43E+01	4.02E+01	-9.20
650	1500	7.98E+01	7.37E+01	-7.54
650	2000	1.22E+02	1.18E+02	-3.23
1500	0	4.99E+01	4.11E+01	-17.60
1500	2000	3.92E+02	3.82E+02	-2.76
1500	4000	1.08E+03	1.09E+03	1.27
1500	6000	2.16E+03	2.19E+03	1.28
2500	0	2.00E+02	1.79E+02	-10.23
2500	2000	9.25E+02	9.01E+02	-2.63
2500	4000	2.23E+03	2.24E+03	0.61
2500	6000	4.13E+03	4.22E+03	2.07
				Avg.: -0.83 Max: 47.51 Min: -17.60

Table 4.7: Partial factorization time estimations for executing MKL kernels with four threads.

4.4.2 Serial Factorization Time

Although the number of flop required for partial factorization dominates the total operation count for numerical factorization, the execution time of the other components can be significant. This is due to the relatively slow speeds of the components other than the partial factorization operations. If optimized BLAS3 kernels are used for the partial factorization, then the partial factorization is performed at a speed close to the machine peak. On the other hand, the speed for finite element and update matrix assembly

operations are significantly slower than the partial factorization speeds since the assembly operations are mainly memory-bound operation, for which obtaining speeds close to the machine peak is difficult due to limited opportunities to exploit data locality. The total factorization times are underestimated if we merely consider the partial factorization times. This is especially true for problems with large non-zero/flop ratios. For these problems, the number of assembly operations relative to flop is high. Figure 4.15 shows the execution time of the different components of the multifrontal factorization for the benchmark suite of 40 test problems. The execution times given in Figure 4.15 are normalized according to the total factorization time. As shown in Figure 4.15, the time spent for handling the update matrices may be a significant portion of the overall factorization time for some test problems. The time spent in update matrix operations is especially high for small 2D problems. For larger problems and 3D problems, the time spent for handling update matrices corresponds to a smaller portion of the overall factorization time.

Figure 4.15 also shows the time spent in the assembly of FE matrices in terms of the overall factorization time. Similar to the time spent for the update matrices, the time spent in the assembly of FE matrices can be significant for the smaller 2D problems (it is larger than 10% of the total factorization time for Models 1&2). The time spent in FE matrix operations, update matrix operations, and partial factorization subroutines adds up to most of the factorization time. The remaining code takes less than 1% of the total factorization time for all test problems in the benchmark suite.

The partial factorization operations required for factorization of a test problem is simulated in order to experimentally determine the time spent in the MKL subroutines. Partial factorization simulations are performed for the dense matrix sizes same as the frontal matrix sizes corresponding to an assembly tree. MKL time shown in Figure 4.15 is the simulated partial factorization times. The simulated partial factorization time is expected to match with the estimated MKL times, which are estimated based on the

approximation method described in the previous section. As shown in Figure 4.15, the partial factorization time estimations are usually in accordance with the actual partial factorization times. If we sum the simulated MKL partial factorization times and the time spent in the assembly operations for FE and update matrices, the sum corresponds to a fraction of the actual factorization times as shown in Figure 4.15. The main reason for that is the simulated partial factorization times is an optimistic measure of the time spent in the partial factorization subroutines during the actual factorization. For the simulated partial factorization, the partial factorization is repeatedly performed on frontal matrices that occupy the same memory location. Therefore, there is mainly compulsory cache misses for the simulated partial factorizations. A compulsory cache miss is a failed attempt to read data from the cache for the first reference to a memory location. However, there is also capacity misses and conflict misses during the actual numerical factorization caused by the entries of the update matrices and FE matrices. Those cache misses are failed attempt to read a data from the cache caused by the replacement of the referenced data with some other data. During the actual factorization, the frontal matrix entries are replaced with the entries of the update matrix stack and FE matrices. On the other hand, for the simulated partial factorization, there is no update matrix stack and FE matrices to replace the entries of the frontal matrix. Therefore, the simulated partial factorizations have smaller execution time compared to the actual time for the partial factorization of the frontal matrices. We assume that the partial factorization is performed 10% less efficient than the optimistic MKL predictions. This is to close the performance gap between the simulated partial factorization times and the actual factorization times shown in Figure 4.15.

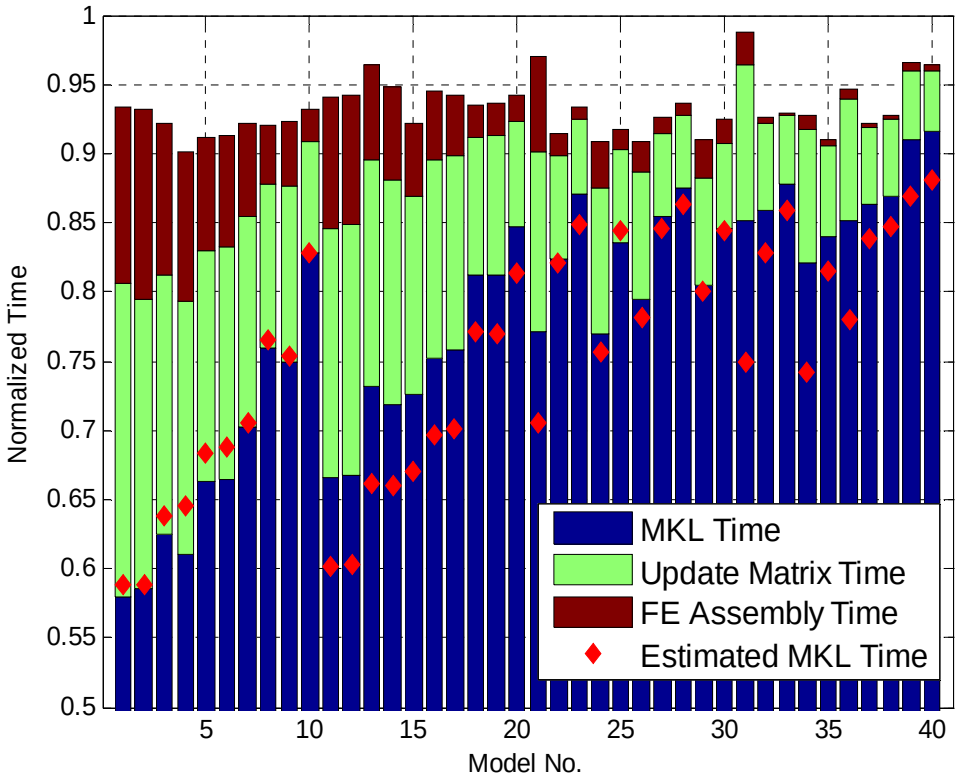


Figure 4.15: The execution time of different components of the solver normalized according to the total factorization time. The plot is for the benchmark suite of 40 test problems (HMETIS)

Next, we calculate the average speed of the update matrix operations and FE assembly operations by using the execution time and number of floating point operations required for these operations. Figure 4.16 shows the average speed of the FE and update matrix operations in terms of flop/sec. Our performance model for the numerical assumes a constant speed for the FE and update matrix operations. According to the results shown in Figure 4.16, we choose a constant speed of 0.04 GFlop/sec for the FE assembly operations and 0.32 GFlop/sec for the update matrix operations. However, neither operation has a constant speed in reality. For example, for smaller 2D problems, the speed of update operations is slow as it is shown in Figure 4.16 for Model 1&2. The size of the frontal matrices is typically small for the small problems. Consequently, the node blocks are small for small frontal matrices and the assembly of the update operations is

slower for smaller node blocks. Therefore, a performance model that considers speed variation of the FE and update matrix operations will yield more realistic factorization time estimations.

In our performance model, we calculate the estimated overall factorization time by summing up the time spent for each assembly tree node. For each assembly tree node, the factorization time is estimated by summing up the estimated partial factorization time, estimated update matrix operations time and estimated FE assembly time. Figure 4.17 shows the estimated factorization times normalized according to the actual factorization times for the same benchmark suite.

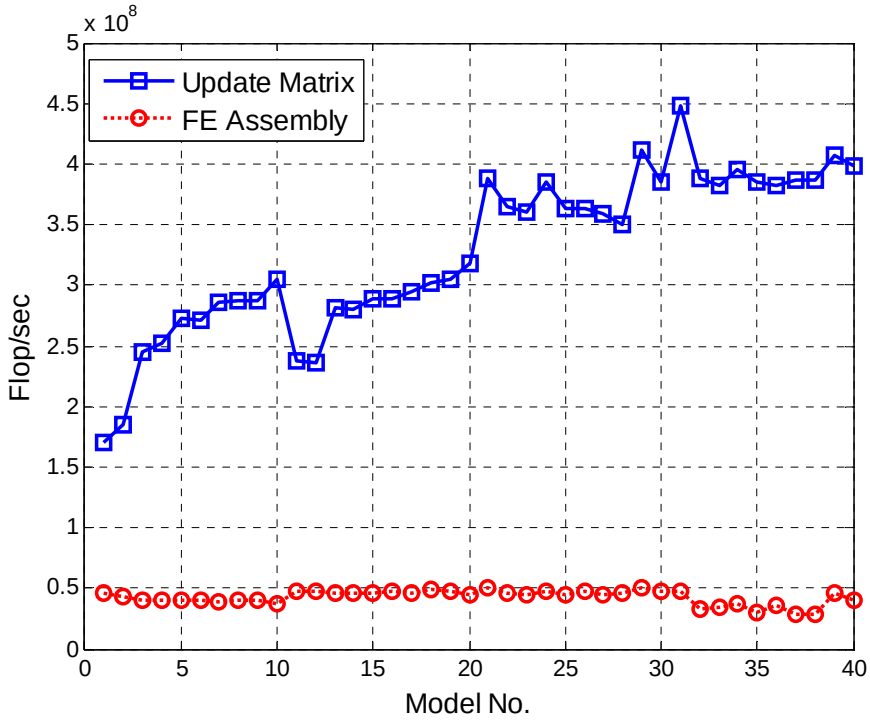


Figure 4.16: The average speed of the update matrix and FE assembly operations for the benchmark suite of 40 test problems (HMETIS)

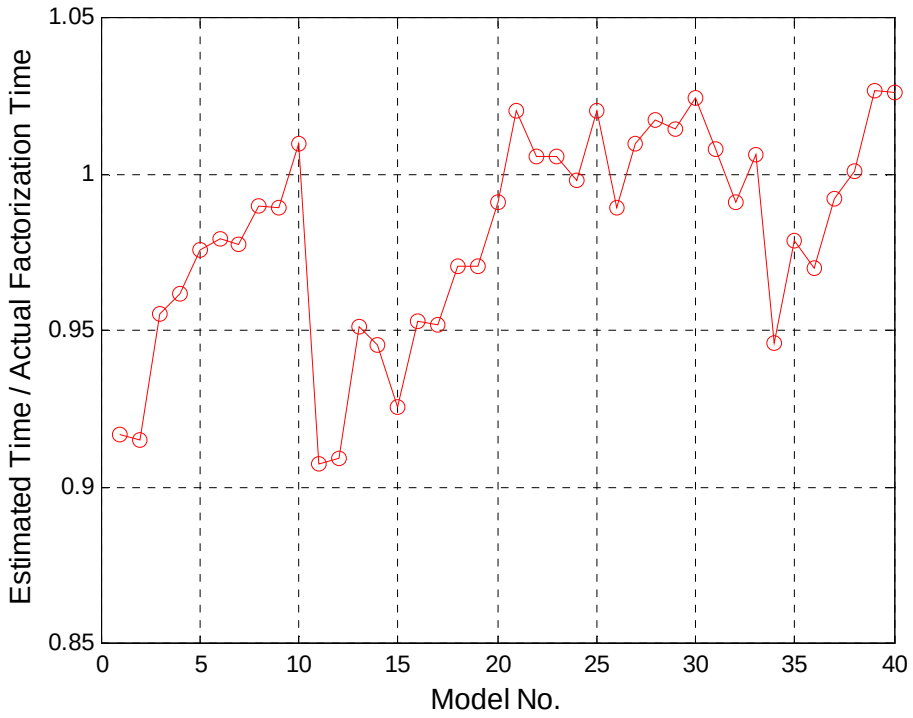


Figure 4.17: Factorization time estimations normalized according to the actual factorization times for the benchmark suite of 40 test problems (HMETIS)

The factorization time estimations can be used to choose among alternative pivot-orderings found in the preprocessing phase of the solver. Table 4.8 shows the estimated and actual factorization times with alternative matrix ordering programs for the test problem f75×150×5. The operation counts and solution times are also given in Table 4.8. As shown in Table 4.8, the estimated factorization times are a better indicator for the relative factorization performance compared to the operation counts for the numerical factorization. Among five matrix ordering programs shown in Table 4.8, AMF yields the smallest operation count (299.87 GFlops). Although the operation count for HMETIS is larger than the one for AMF (356.89/299.87=1.19×), HMETIS yields a better factorization time (63.2/48.3=1.31×). This is due to the large number of update operations for the factorization with AMF matrix ordering program (3.42 GFlop compared to 0.96 Gflop of HMETIS). The developed performance model incorporates time spent in the update matrix operations and successfully predicts the factorization

performance of the matrix ordering programs as shown in Table 4.8. HMETIS yields the most favorable estimated factorization time and factorization time. The solution performance of different matrix ordering programs is similar to their factorization performance as shown in Table 4.8.

Matrix Ordering Program	Factorization Operation Count (GFlop)	Estimated Factorization Time (sec)	Factorization Time (sec)	Solution Time with 100 RHS (sec)	Update Matrix Operation Count (GFlop)
AMF	***299.87***	56.7	63.2	14.5	3.42
AMD	489.88	71.1	70.1	14.5	1.17
MMD	424.11	63.2	63.0	14.4	1.24
HMETIS	330.09	***48.9***	***48.3***	***12.3***	***0.96***
HAMF	356.89	55.3	55.4	13.6	1.51

Table 4.8: Choosing the best pivot-ordering among alternatives based on the estimated factorization time (f75×150×5)

4.4.3 Multithreaded Factorization Time

Subtrees of the assembly tree are assigned to threads if multiple threads are used for the factorization. An example subtree with thread mapping is shown in Figure 4.18 for an example assembly tree. Here, the partial factorization for the subtrees is performed by using serial BLAS/LAPACK kernels. Therefore, the execution time for the subtrees is estimated by summing up the estimated partial factorization times of the tree nodes as it is described in the previous section. The parallel factorization time for the subtrees is determined by the subtree with the largest serial execution time.

For the tree nodes at a higher level than the subtree nodes, the factorization is performed using all available threads by using the multithreaded BLAS/LAPACK kernels. The speedup approximation found in Chapter 4.4.1 is used to determine the multithreaded partial factorization times. The speedup is not applied to the time required for copying the update matrices to the stack since this portion of the code is executed serially. Although multithreaded BLAS kernels are used for the assembly of the update matrices to the frontal matrix, we neglect the speedup for this operation too. The pseudo

code for estimating the multithreaded factorization time is given in Figure 4.19. In Figure 4.19, the subtree factorization times are found in the first loop. The last loop in the code calculates the partial factorization time for the tree nodes above the subtrees (nodes 25, 26, and 27 in Figure 4.18). For calculating the total factorization time, the execution time for the top level nodes is added to the maximum subtree factorization time.

The estimated multithreaded factorization times are usually smaller than the actual factorization times. In other words, the multithreaded factorization performs worse than the expectations. The underestimation of the multithreaded factorization time is explained in Chapter 7.

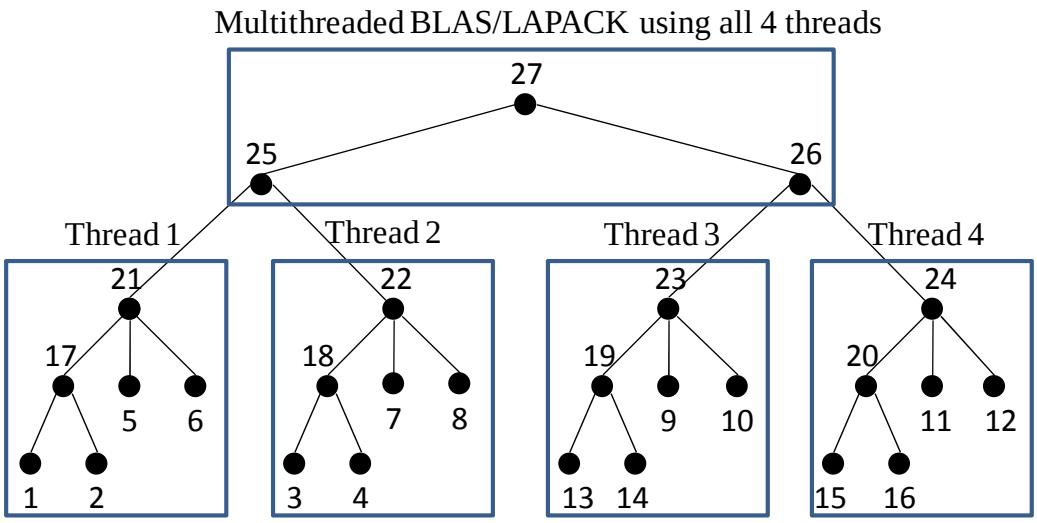


Figure 4.18: Four thread factorization of an example assembly tree.

```
The function that calculates the estimated parallel factorization time
INPUT:
  subtree_nodes: arrays storing the subtree nodes that can be processed independently
  top_level_nodes: the tree nodes above the subtree nodes
RETURNS:
  estimated parallel execution time for given subtree_nodes and top_level_nodes.
function estimate_exe_time(subtree_nodes, top_level_nodes)
i := 0
for i := 0 to number of subtrees
    subtree_exe_times[i] := 0
    for each subtree_node in subtree_nodes[i]
        subtree_exe_times[i] := subtree_exe_times[i] + estimate_facto(subtree_node)
                                estimate_update(subtree_node) + estimate_FE_assembly(subtree_node)
    end for
end for
subtree_exe_time := max(sub_tree_exe_times)
top_exe_time := 0
for each tree_node in top_level_nodes
    flop := flop required for the factorization of the tree_node
    speedup := estimate_speedup(flop)
    top_exe_time := top_exe_time + estimate_facto(tree_node)/speedup +
                    estimate_update(tree_node) + estimate_FE_assembly(subtree_node)
end for
return subtree_exe_time + top_exe_time
```

Figure 4.19: The pseudo code for estimating the multithreaded factorization time.

4.5 Mapping Algorithm

Two levels of parallelism are exploited in the SES direct sparse solver: tree-level parallelism and dense matrix level parallelism. The tree-level parallelism is performing the partial factorization in parallel for the tree nodes between which there is no dependency. The dense matrix level parallelism is exploited by the use of the multithreaded MKL linear algebra kernels. If we exploit only the tree level parallelism, then this is not a scalable approach since the majority of the computations are performed at the top three levels of the assembly tree [142-143]. Therefore, exploiting parallelism at the top level assembly tree nodes is essential for a scalable direct sparse solver. The existing parallel sparse solvers usually exploit the dense matrix level parallelism. For example, the MUMPS sparse solver exploits the parallelism at high-level tree nodes by computing the Schur complement in parallel for large frontal matrices [143-144]. In addition, the factorization of the root node is performed in parallel using the

ScaLAPACK kernels. The root node is partitioned and distributed among the processors in 2D block cyclic distribution [145]. The PARDISO [97] and PASTIX [94] solvers exploits further parallelism by splitting large supernodes close to the root of the assembly tree. Recently, Hogg et al. [146] proposed a DAG driven scheme for parallel sparse Cholesky Decomposition on multi-core processors. The scheme is similar to the tiled algorithms proposed by Buttari et al. [30] and it schedules the factorization tasks based on a dependency graph for the tasks associated with dense matrix blocks. This approach is typically more scalable compared to merely relying on the multithreaded performance of the BLAS3 kernels [30].

In order to exploit the tree-level parallelism, the subtrees that can be processed independently can be found by using the mapping algorithm proposed by Geist and Ng [147]. In this scheme, the subtrees are explored in the decreasing order of their workloads and a subtree is assigned to the processor with the lightest workload. This is called bin-packing heuristic, where subtrees are assigned to the bins with the lightest workloads. The bin-packing is repeated until the load imbalance ratio of the bins is smaller than a user-specified tolerance value.

After the subtrees are found with the bin-packing heuristic, the tree nodes above the subtrees remain to be mapped to the processors. Pothen and Sun [148] proposed a mapping algorithm that maps the tree nodes above the subtrees by considering the communication costs. In the current implementation of the SES solver package, the high-level tree nodes are processed by the main thread only, which employs the multithreaded BLAS/LAPACK kernels for the partial factorization. In other words, the tree-level parallelism is not exploited for the high-level tree nodes. This gives satisfactory speedups for the SMP multi-core processors. However, for NUMA multi-core architectures a mapping algorithm that considers the data locality of the high-level tree nodes may be required for high-performance. The mapping algorithm proposed by Pothen and Sun [148] can be used for this purpose.

Parallel solvers either use a static scheduling, such as PASTIX, or a combination of dynamic and static scheduling, such as MUMPS and PARDISO. The dynamic scheduling is especially useful when the workload cannot be predicted accurately prior to the numerical factorization due to the delayed factorization of the supernodes for numerical stability concerns. On the other hand, if the workload predictions are accurate and the workload of the processors is balanced, the static scheduling yields a good performance (for example, see Kurc et al. [81] and Hennon et al. [94]). SES solver package determines the tasks that will be scheduled in the processors statically by using the partial factorization time predictions. This typically assures a balanced workload among the threads. Previous sections explain how the factorization times are predicted by using the experimental execution times for the BLAS/LAPACK subroutines.

The SES scheduling algorithm aims to exploit two-levels of parallelism in an optimal fashion so that the parallel factorization time is minimized. We determine when to switch to tree-level parallelism from dense-matrix level parallelism based on the estimated factorization times. The mapping algorithm is iterative and it searches for a subtree to thread mapping that minimizes the estimated parallel execution time. The iteration starts at the head node of the assembly tree. At this point, the entire tree is considered as a subtree and it is processed by a single thread. At each iteration step, independent subtrees that can be processed in parallel are found by performing a breadth first search. The tree nodes are assigned to the processors in a decreasing order of the estimated subtree factorization times in a cyclic fashion. We use a priority queue to hold the independent subtrees and to assign the subtrees to the processers in the decreasing order of the subtree processing time. The priority queue is also used for performing the breadth first search. Figure 4.20 shows the successive steps of the search performed within the mapping algorithm for an example assembly tree. The serial execution time estimations for the tree nodes are also shown in Figure 4.20. At each step of the search, the parallel factorization time is predicted by using our performance model. The search is

continued until the parallel execution time stops decreasing. The pseudo code for the mapping algorithm is given in Figure 4.21.

The mapping algorithm may assign multiple subtrees to a thread. The numerical factorization waits until all threads finish their work on the subtrees assigned to them. Then, the high-level tree nodes are processed by using multithreaded BLAS/LAPACK kernels with maximum available number of threads. As it is stated earlier, we do not exploit tree-level parallelism for the tree nodes above the subtrees. If the synchronizations within the BLAS/LAPACK kernels are not considered, the factorization using the subtree to thread mapping requires a single synchronization between the threads. The only synchronization point is before starting to process the first high-level assembly tree node. The subtree to thread mapping found for the factorization is also used for forward elimination and back substitution. For the back substitution, the tasks are processed in the reverse order of the factorization. The use of the same mapping simplifies the implementation for the forward elimination and back substitution. The items and data structures that are created for the numerical factorization can also be used in the triangular solution such as:

- Synchronization constructs for the threads
- The tree-traversal found in the symbolic factorization
- The factors calculated by threads
- The frontal matrices
- The subtree to thread mapping

The main disadvantage of using the same mapping is that it may lead to workload imbalances in forward elimination and back substitution phases since the relative execution times of the factorization and triangular solution may not be similar for two different assembly tree nodes.

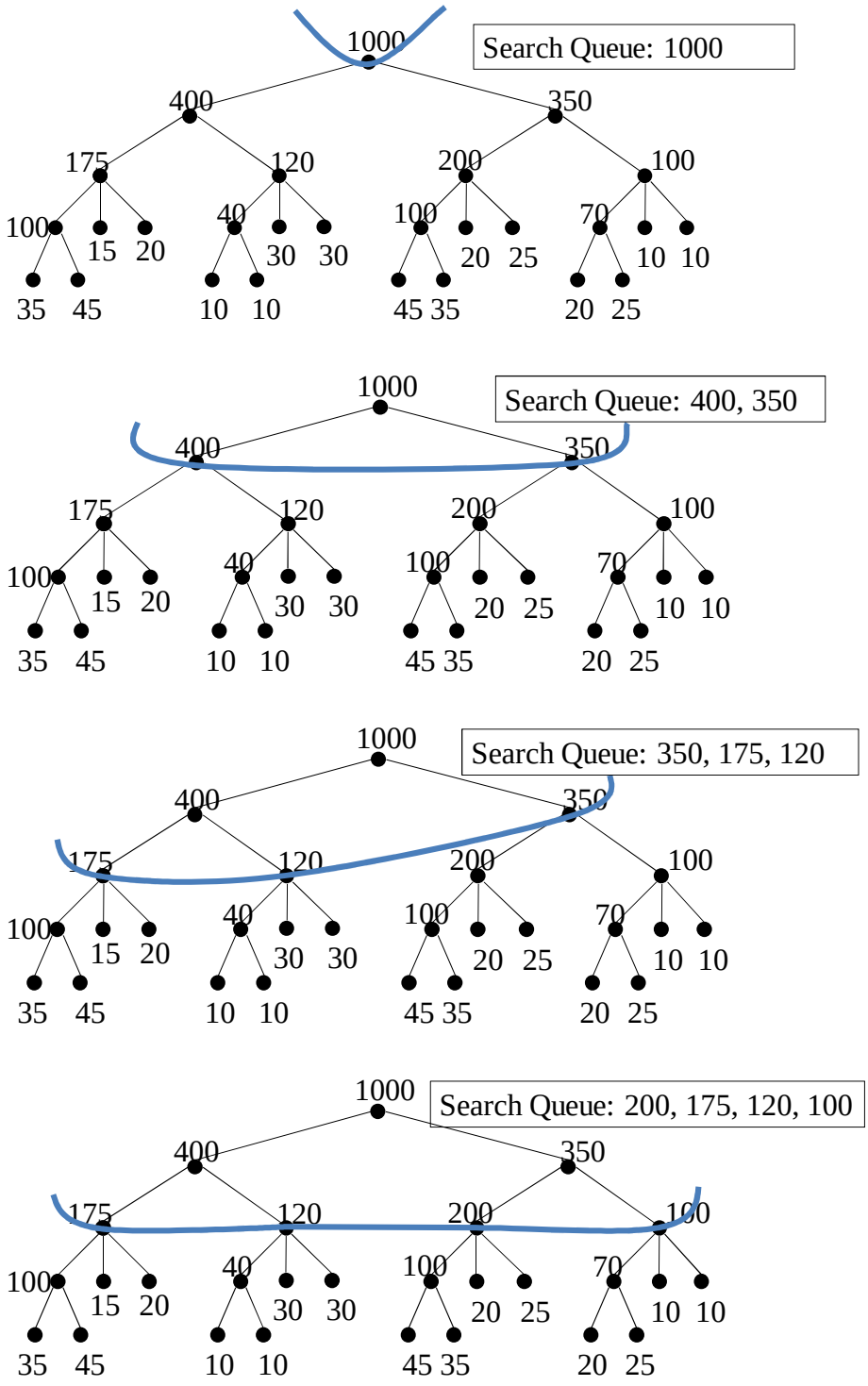


Figure 4.20: The search for the independent subtrees that can be processed in parallel.

INPUT:
root_node: the root node of the assembly tree
n: the number of additional iterations after an increase in estimated factorization time is detected

OUTPUT:
top_level_nodes: the tree nodes that are processed with multithreaded BLAS/LAPACK
subtree_nodes: arrays storing subtree nodes that can be processed independently
parallel_facto_time: parallel factorization time

```
initialize the top_level_nodes and subtree_nodes to an empty set
initialize the priority_queue with the root_node
counter := 0
while counter < n
    removed_node := extract the first element in the priority_queue
    (a) add the children of the removed_node to the priority_queue
    (b) add the removed_node to the top_level_nodes_candidate
    (c) assign the nodes in the priority_queue to the subtree_nodes_candidate in decreasing
        order of the tree execution times in a cyclic distribution
    /* use the estimate_time function given in Figure 4.19 to predict execution time */
    time := estimate_time(top_level_nodes_candidate, subtree_nodes_candidate)

    if time > best_time
        counter := counter + 1
    else
        best_time := time
        subtree_nodes := subtree_nodes_candidate
        top_level_nodes := top_level_nodes_candidate
        counter := 0
    end if

end while
parallel_facto_time := best_time
```

Figure 4.21: The pseudo code for subtree to thread mapping algorithm.

4.6 Symbolic Factorization

The numerical factorization and triangular solution are simulated in the symbolic factorization in order to determine the memory requirements and data structures for the numerical solution. First, an assembly order is found for each thread based on the subtrees found in the mapping algorithm. We employ postorder tree traversals for the subtrees and the assembly tree nodes above the subtrees. Current implementation of the SES solver package does not employ an optimal postorder traversal to minimize the active memory requirement. In future versions of the solver package, the memory

minimizing schemes developed by Liu [55] and Guermouche and L'excellent [54] will be employed.

Once the ordering of the partial factorization operations is found for threads, memory required for the frontal matrix, update matrix stack, RHS vectors, and factors is determined. The memory requirements are calculated in parallel for the data structures corresponding to each thread. Finally, the local indices are found for frontal matrices.

CHAPTER 5

FACTORIZATION

&

TRIANGULAR SOLUTION PHASES

Once a strategy for the parallel numerical solution is determined in the analysis phase, the stiffness matrix is factorized using the multifrontal method [51]. The unknowns are then found by performing forward elimination and back substitution. This chapter discusses the numerical factorization and triangular solution algorithms and their implementation.

5.1 Numerical Factorization

The multifrontal method [51] is employed for the numerical factorization. The multifrontal method is the generalization of Irons' frontal method [45] which allows the use of multiple frontal matrices. In the multifrontal method, the numerical factorization is reduced to a series of partial factorization operations on dense frontal matrices. The partial factorization on a frontal matrix is performed by finding the factors for the fully assembled dofs and finding the Schur complement for partially assembled dofs. The fully assembled dofs are the dofs for which all connected elements have been assembled in the frontal matrix. They are also called the eliminated dofs. On the other hand, the partially assembled dofs are the dofs for which all connected elements are not assembled. They are also called remaining dofs. The partial factorization is similar to the condensation of a frontal matrix where the condensed dofs are fully assembled dofs and the remaining dofs are partially assembled dofs. Typically, we exploit the supernodes and there are multiple dofs eliminated at a frontal matrix. Therefore, we can use a blocked form of the partial factorization that can be written as [52]:

$$\begin{bmatrix} \mathbf{B} & \mathbf{V}^{\mathrm{T}} \\ \mathbf{V} & \mathbf{C} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{\mathrm{B}} & \mathbf{0} \\ \mathbf{V}\mathbf{L}_{\mathrm{B}}^{-\mathrm{T}} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{C} - \mathbf{V}\mathbf{B}^{-1}\mathbf{V}^{\mathbf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{\mathrm{B}}^{\mathrm{T}} & \mathbf{L}_{\mathrm{B}}^{-1}\mathbf{V}^{\mathrm{T}} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \tag{5.1}$$

The matrix on the left hand side of Equation 5.1 is called the frontal matrix. The frontal matrix is composed of the dense matrices $\mathbf{B}_{ne \times ne}$, $\mathbf{V}_{nr \times ne}$, and $\mathbf{C}_{nr \times nr}$ where *ne* and *nr* are the number of eliminated and remaining dofs respectively. **B** is a square matrix corresponding to the eliminated dofs, **C** is a square matrix corresponding to the remaining dofs, and **V** is a rectangular matrix which couples the eliminated and remaining dofs. After the partial factorization is completed, **B** stores the diagonal factors $\mathbf{L}_{\mathrm{B}}$, **V** stores the off-diagonal factors $\mathbf{V}\mathbf{L}_{\mathrm{B}}^{-\mathrm{T}}$, and **C** stores the Schur complement $\mathbf{C}–\mathbf{V}\mathbf{B}^{-1}\mathbf{V}^{\mathrm{T}}$.

The LAPACK and BLAS3 subroutines are used for performing the partial factorization shown in Equation 5.1. Symmetric diagonal factors, $\mathbf{L}_{\mathrm{B}}$, in Equation 5.1 are computed with a call to the dpotrf function in LAPACK. This function computes the Cholesky factors of a symmetric Hermitian matrix. An efficient implementation of this function divides matrix **B** into blocks sized such that the blocks fit into the memory cache of the machine [149]. Once the diagonal factors, $\mathbf{L}_{\mathrm{B}}$, are found, the off-diagonal factors, $\mathbf{V}\mathbf{L}_{\mathrm{B}}^{-\mathrm{T}}$, are computed using the dtrsm function in BLAS3. The dtrsm function finds the off-diagonal factors by solving the following equation for $\mathbf{L}_{\mathrm{off}}$:

$$\mathbf{L}_{\mathrm{off}}\mathbf{L}_{\mathrm{B}}^{\mathrm{T}} = \mathbf{V} \tag{5.2}$$

From the above equation, $\mathbf{L}_{\mathrm{off}}$ can be written as:

$$\mathbf{L}_{\mathrm{off}} = \mathbf{V}\mathbf{L}_{\mathrm{B}}^{-\mathrm{T}}$$

Finally, the Schur complement, **S**, can be found by:

$$\mathbf{S} = \mathbf{C} - \mathbf{V}\mathbf{B}^{-1}\mathbf{V}^{\mathrm{T}} = \mathbf{C} - \mathbf{L}_{\mathrm{off}}\mathbf{L}_{\mathrm{off}}^{\mathrm{T}} \tag{5.3}$$

where $\mathbf{L}_{\mathrm{off}}$, off diagonal factors, is substituted for $\mathbf{V}\mathbf{L}_{\mathrm{B}}^{-\mathrm{T}}$. The dsyrk function in BLAS3 is used for the rank *nc* update of the **C** matrix given in Equation 5.3.

The matrices **B** and **C** are symmetric and a packed storage scheme can be employed for their storage. In packed storage scheme, either the upper or lower diagonal

entries of a dense matrix are stored. However, **B** and **C** are stored as if they are full matrices since there is no BLAS3 kernel for rank-k update that uses the packed storage scheme. Additionally, the LAPACK implementation in the Intel's MKL library performs better when a full matrix storage scheme is employed [140].

Next, we give the number of floating point operations required for the partial factorization of a frontal matrix with *ne* eliminated variables and *nr* remaining variables. For dense frontal matrices, the number of floating point operations required for calculating the diagonal factors $\mathbf{L}_B$ is given as:

$$\sum_{i=1}^{ne} i + \sum_{i=1}^{ne} i(i-1) = \frac{ne^3}{3} + \frac{ne^2}{2} + \frac{ne}{6} \tag{5.4}$$

where the first term on the left hand side is number of floating point operations required to find the factors for the current pivot. The second term is number of update operations performed for the columns on the right side of the pivot column.

The off-diagonal factors $\mathbf{L}_{off}$ is computed by performing a triangular solution with *nr* RHS vectors. The number of floating point operations required for the $\mathbf{L}_{off}$ is given as:

$$nr\sum_{i=1}^{ne}\left(2(i-1)+1\right) = ne^2 nr \tag{5.5}$$

Finally, the number of floating point operations required for the calculation of the **S** in Equation 5.3 is given as:

$$\sum_{i=1}^{nr}\sum_{j=i}^{nr} 2ne = ne \cdot nr^2 + ne \cdot nr \tag{5.6}$$

where the nested summation represents the number of entries in the **S**. The 2*ne* term inside the summation is the number of floating point operations required for calculating each entry in the **S.** The total number of floating point operations required for the partial factorization is the summation of the Equations 5.4-6 and is given below:

$$\frac{ne^3}{3} + ne^2 nr + \frac{ne^2}{2} + ne \cdot nr^2 + ne \cdot nr + \frac{ne}{6} \tag{5.7}$$

The majority of arithmetic operations are to compute $\mathbf{L}_B$ if *ne* is large relative to *nr*. On the other hand, if *nr* is larger than *ne*, the majority of operations are to compute **S**. Table 5.1 gives the operation counts for computing $\mathbf{L}_B$, $\mathbf{L}_{off}$ and **S** in terms of the total operation count required for partial factorization. As shown in Table 5.1, the majority of the operations are for computing **S** if *ne*/*nr* is smaller than 0.8. As the ratio *ne*/*nr* increases, the operation count ratio for computing **S** decreases.

ne/nr	$\mathbf{L}_B$ Ratio	$\mathbf{L}_{off}$ Ratio	**S** Ratio
0.2	0.011	0.165	0.824
0.4	0.037	0.275	0.688
0.6	0.07	0.349	0.582
0.8	0.106	0.397	0.497
1	0.143	0.428	0.429
1.2	0.179	0.447	0.373
1.4	0.214	0.458	0.328
1.6	0.247	0.463	0.29
1.8	0.278	0.464	0.258
2	0.308	0.461	0.231
3	0.429	0.428	0.143
4	0.516	0.387	0.097
5	0.581	0.349	0.07

Table 5.1: For frontal matrices with different *ne*/*nr* ratios, the operation counts for computing $\mathbf{L}_B$, $\mathbf{L}_{off}$ and **S** given in terms of total operation count for the partial factorization

For four example test problems, Table 5.2 shows the mean *ne*/*nr* ratios for the assembly tree constructed by using the HMETIS matrix ordering. As shown in Table 5.2, mean *ne*/*nr* ratios are smaller than 0.4 for 2D example problems and they are smaller than 0.2 for 3D example test problems. For the assembly tree of the example problems, the *ne*/*nr* ratio is usually larger than the mean ratio for the tree nodes close to the root. The *ne*/*nr* ratio is infinity for the root tree node since there are no remaining variables at the

root. The root node is not included in the statistics shown in Table 5.2. Table 5.3 shows the total operation counts for the factorization of the assembly tree constructed with HMETIS ordering. As shown in Table 5.3, the majority of the factorization operations are for computing the Schur complement **S**. This is expected since the mean for *ne*/*nr* ratios are small as shown in Table 5.2. For the test problems shown in Table 5.3, about 70% of the operations are for computing the **S** matrices. The operation counts for computing $\mathbf{L}_B$ at the root node is larger than the sum of the operation counts for computing $\mathbf{L}_B$ at the remaining tree nodes. For 3D problems shown in Table 5.3, the operation counts for computing $\mathbf{L}_B$ at the root node are more than 85% of the sum operation counts for computing $\mathbf{L}_B$ at all tree nodes including the root tree node.

Model Name	Mean *ne*/*nr*	Standard Deviation *ne*/*nr*	Maximum *ne*/*nr*	Minimum *ne*/*nr*
q500×500	0.302	0.170	1.5	0.008
f500×500	0.376	0.187	1.8	0.002
s30×30×30	0.098	0.053	0.5	0.038
f30×30×30	0.172	0.058	0.625	0.002

Table 5.2: Assembly tree statistics for the example test problems

Model Name	Operation Count $\mathbf{L}_B$ (GFlop)	Operation Count $\mathbf{L}_{off}$ (GFlop)	Operation Count **S** (GFlop)	Total Operation Count (GFlop)
q500×500	1.726	4.548	15.113	21.385
f500×500	4.806	9.661	31.778	46.245
s30×30×30	25.968	22.883	91.698	140.549
f30×30×30	51.683	72.750	314.333	438.767

Table 5.3: Arithmetic operation counts for the factorization of the example test problems

A partial factorization is performed for each frontal matrix corresponding to an assembly tree node. The dependencies between the partial factorization tasks are shown in Figure 5.1 for an example assembly tree. For the factorization, the tree nodes are visited in a postorder tree traversal, for which the children of an assembly tree node are visited before the parent node. A single active frontal matrix is used for the numerical factorization for the serial factorization. For multithreaded factorization, the active frontal matrices are as many as the number of threads used for the factorization if the tree-level parallelism is exploited. A stack data structure can be used to store **S** since a postorder tree traversal is employed. The stack data structure that stores **S** matrices is referred to as update matrix stack. Prior to a partial factorization, the update matrices of the children are popped from the stack and assembled into the frontal matrix.

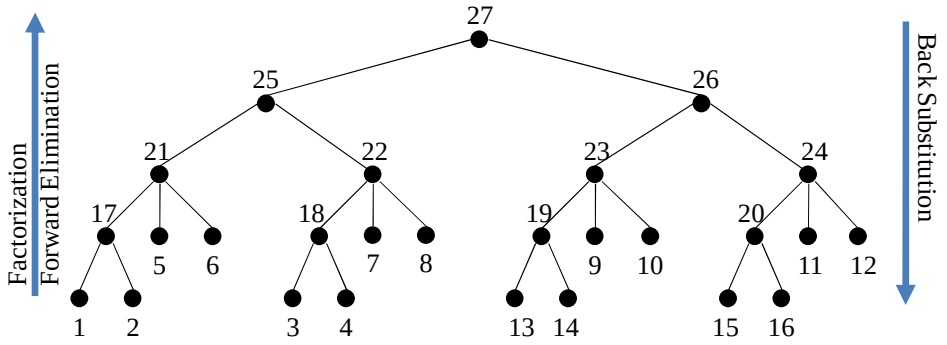


Figure 5.1: Direction of the dependencies between the factorization and triangular solution tasks for the example assembly tree.

For multithreaded multifrontal factorization, each thread has its own frontal matrix and update matrix stack. The size and structure of the update matrix stacks are determined in the analysis phase of the solver package. Similarly, the size of the frontal matrix is determined in the analysis phase during the symbolic factorization. Once the total memory requirements are determined, memory is allocated for the frontal matrices

and update matrix stacks prior to the numerical factorization. There is no need for the memory reallocation since a static scheduling is employed and the operations performed by each thread are predetermined in the symbolic factorization phase. We also avoid the dynamic memory allocations during the numerical factorization. As dynamic memory allocations typically require system level synchronization of memory, a large number of dynamic memory allocations for small blocks of memory is typically inefficient compared to allocating large memory blocks at the same time.

Figure 5.2 shows the pseudo code for multithreaded factorization. This code is executed in parallel by multiple threads. Each thread works on an exclusive set of tree nodes, which is given as an input to the code shown in Figure 5.2. The tree nodes are held in an array and they are stored according to a post-traversal tree node ordering, which is found in the analysis phase. Before processing each node in the array, first, it is checked whether synchronization is required for the partial factorization of the current tree node. If the partial factorization of a frontal matrix must wait for other threads, the frontal matrix is constructed after the dependent threads have finished. If the frontal matrix of a child tree node is processed by the current thread then the update matrix of the child is popped from the update matrix stack of the current thread. Otherwise, a search is required to make sure that we assemble the correct update matrix. Our implementation does not require synchronization constructs for accesses to the update matrix stack. The synchronizations shown in Figure 5.2 guarantee that only a single thread accesses an update matrix stack at a time. The construction of the frontal matrix is finalized by assembling all finite elements that contains the fully assembled dofs. After the frontal matrix is constructed, a partial factorization is performed by using the LAPACK/BLAS subroutines described previously. Once the factors are found, they are stored for later use in the triangular solution. Once **S** is found for the frontal matrix, it is pushed to the update matrix stack since it will be required during the partial factorization of the parent assembly tree node.

There are two important requirements that must be enforced for the correct execution of the multifrontal factorization code given in Figure 5.2. First, the partial factorization of high-level tree nodes should be in the reverse order of the partial factorization of the subtree tree nodes for the main thread. This is to ensure that the stack data structure can be used for the tree nodes above the subtrees. Second, for a parent assembly tree node, ordering of its children nodes should be in accordance with the ordering of the tree nodes that is given as input. Otherwise, pops from the update matrix stack may bring an update matrix that belongs to a child node that is a sibling of the desired child node. The analysis phase finds postorder tree traversals for the threads that satisfy these two requirements.

The multithreaded multifrontal factorization algorithm

INPUT:

current_thread_id: the thread ID currently executing the function
tree_nodes: the assembly tree nodes assigned to the current thread (in postorder traversal)
$\mathbf{L}_B$, $\mathbf{L}_{off}$, **S**: the frontal matrix blocks for the current thread
update_stacks: the update stacks for all threads
max_thread_num: the number of threads used for the numerical factorization

OUTPUT:

factors: the factors found by the multifrontal factorization, each thread has its own copy.

```
/* traverse all nodes assigned to the thread */
for each tree_node in tree_nodes
    /* wait until all children nodes that the tree node depends are processed */
    if requires_wait(tree_node) then wait_for_threads(tree_node) end if
    /* traverse all children nodes of the current tree node */
    for each child_node of the tree_node
        /* get the thread id that processed the children node. */
        thread_id := get_thread(child_node)
        /* pop the update matrix from the corresponding update stack. This is S computed at the
        children tree node*/
        if thread_id = current_thread_id then
            update_matrix:=pop(update_stacks[thread_id])
        otherwise
            /* find the child node's update matrix in the update matrix stack */
            update_matrix:=find(child_node, update_stacks[thread_id])
        end if
        assemble(update_matrix, L_B)
        assemble(update_matrix, L_off)
        assemble(update_matrix, S)
    end for
    /* traverse FE's for the tree node*/
    for each element_matrix stored in the tree_node
        assemble(element_matrix, L_B)
        assemble(element_matrix, L_off)
        assemble(element_matrix, S)
    end for
    /* if it is a high-level tree node then use all threads available for partial factorization */
    if is_subtree_node(tree_node) then set_blas_lapack_thread_num(1)
    otherwise set_blas_lapack_thread_num(max_thread_num) end if
    /* make the LAPACK and BLAS calls required for the partial factorization */
    L_B:= dpotrf(L_B)
    L_off:=dtrsm(L_off, L_B)
    S:=dsyrk(L_off, S)
    /* store the results of partial factorization*/
    store_diagonal(tree_node, L_B, factors)
    store_off_diagonal(tree_node, L_off', factors)
    push(S, update_stacks[current_thread_id])
    /* notify the threads that waits for this tree node */
    if requires_notification(tree_node) then notify_threads(tree_node) end if
end for
```

Figure 5.2: The pseudo code for multithreaded numerical factorization algorithm

5.2 Triangular Solution

Once the factors are calculated, the triangular solution can be performed by forward elimination and back substitution. The forward elimination for an assembly tree node can start as soon as the factors corresponding to fully assembled dofs are computed. Alternatively, the factorization and forward elimination can be separated by starting the forward elimination after all factorization steps are completed. The back substitution, on the other hand, must wait until all forward elimination tasks are completed.

5.2.1 Forward Elimination

The efficiency of BLAS3 kernels are extended to the triangular solution with multiple RHS vectors. We employ frontal RHS matrices to perform forward elimination and back substitution operations efficiently on dense matrix blocks. The algorithm is similar to the numerical factorization. However, now, in addition to the frontal matrices that store the factors for the eliminated dofs, we also have a RHS frontal matrix that stores the loads and partial results for the eliminated and remaining dofs. For the forward elimination steps, the frontal matrix and loads on the corresponding dofs are written in a blocked form as follows:

$$\begin{bmatrix} \mathbf{L}_{\mathrm{B}} & 0 \\ \mathbf{L}_{\mathrm{off}} & \mathbf{X} \end{bmatrix} \begin{bmatrix} \mathbf{Y}_{\mathrm{e}} \\ \mathbf{Y}_{\mathrm{r}} \end{bmatrix} = \begin{bmatrix} \mathbf{F}_{\mathrm{e}} \\ \mathbf{F}_{\mathrm{r}} \end{bmatrix} \tag{5.8}$$

where $\mathbf{F}_{\mathrm{e}}$ and $\mathbf{F}_{\mathrm{r}}$ store the loads updated with the partial solution on the eliminated and remaining dofs respectively. The matrices $\mathbf{F}_{\mathrm{e}}$ and $\mathbf{F}_{\mathrm{r}}$ are *ne* by *nrhs* and *nr* by *nrhs* matrices respectively. The right-hand side matrix in Equation 5.8 is referred to as RHS frontal matrix. When the forward elimination steps are complete, $\mathbf{F}_{\mathrm{e}}$ stores $\mathbf{Y}_{\mathrm{e}}$, the results of the forward elimination for the fully assembled dofs. $\mathbf{F}_{\mathrm{r}}$, on the other hand, stores $\mathbf{Y}_{\mathrm{r}}$, the contribution from the current assembly tree node to the forward elimination steps at the parent assembly tree node. Matrix $\mathbf{X}$ in Equation 5.8 is not required for the forward

elimination and back substitution. For forward elimination, the dense matrix operations on the frontal matrix are given as follows:

$$\mathbf{Y}_e = \mathbf{L}_B^{-1}\mathbf{F}_e \tag{5.9}$$

$$\mathbf{F}_r^u = \mathbf{F}_r - \mathbf{L}_{off}\mathbf{Y}_e \tag{5.10}$$

where $\mathbf{Y}_e$ is the results from the forward elimination and $\mathbf{F}_r^u$ is the contribution from the currently processed tree node to the forward elimination steps at the parent tree node. In a way, $\mathbf{F}_r^u$ is similar to **S,** the Schur complement, which is the update matrix for the parent tree node. We do not need extra memory to store $\mathbf{Y}_e$ and $\mathbf{F}_r^u$ since $\mathbf{Y}_e$ and $\mathbf{F}_r^u$ are written on $\mathbf{F}_e$ and $\mathbf{F}_r$ respectively. Before starting the operations on the frontal matrix for another assembly tree node, $\mathbf{Y}_e$ is stored to a separate memory location for a later use in the back substitution.

The BLAS3 dtrsm and dgemm functions are used to perform operations shown in Equations 5.9 and 5.10 respectively. It should be noted that using the suitable BLAS2 kernels will be more efficient if the structure has a small number of loading conditions. In addition, the efficiency of BLAS3 kernels will not shift the extra time spent to store the RHS update matrices $\mathbf{F}_r^u$ for a small number of RHS vectors. Therefore, our approach is especially efficient for the solution of multiple RHS vectors.

Next, we give the number of floating point operations required for the forward elimination steps on the frontal matrix. The dense matrix $\mathbf{Y}_e$ in Equation 5.9 is computed by performing forward elimination on a dense matrices. For a dense square coefficient matrix with the size *ne*, the number of floating point operations required for forward elimination with *nrhs* RHS vectors is given as follows:

$$nrhs\sum_{i=1}^{ne}\left(2(i-1)+1\right)=ne^2 nrhs \tag{5.11}$$

The number of floating point operations required for the operations shown in Equation 5.10 is given as follows:

$$\sum_{i=1}^{nr}\sum_{j=1}^{nrhs} 2ne = 2 \cdot ne \cdot nr \cdot nrhs \tag{5.12}$$

where the term inside the summation represents the number of operation performed to compute each entry of the $\mathbf{F}_r^u$. The total number of floating point operations required for the back substitution is the summation of Equation 5.11-12 and is given as follows:

$$\left(ne^2 + 2 \cdot ne \cdot nr\right) \cdot nrhs \tag{5.13}$$

As shown in the above equation, the operation counts for forward elimination increases linearly as we increase the number of RHS vectors (*nrhs*). A fill-in minimization scheme will also minimize the operation count for forward elimination since a fill-in minimization scheme aims to reduce $ne^2/2 + ne \cdot nr$, the number of nonzero for the pivot columns. For the example test problems, Table 5.4 shows the total operation counts for the forward elimination with 100 RHS. As shown in Table 5.4, the operation count for computing $\mathbf{F}_r^u$ in Equation 5.10 is significantly larger than the operation count for computing $\mathbf{Y}_e$ in Equation 5.9. This is mainly due to small mean *ne*/*nr* ratios of the assembly tree nodes as shown in Table 5.2. If the operation counts in Table 5.3 and Table 5.4 are compared, the total operation count for forward elimination with 100 RHS vectors is comparable to the total operation count for the factorization of 2D example test problems. On the other hand, for 3D test problems, the factorization operation counts are significantly larger than the operation counts for the forward elimination with 100 RHS vectors as shown in the tables. The operation counts for alternative number of RHS vectors can be found by linearly scaling the operation counts given for 100 RHS vectors in Table 5.4.

Model Name	Operation Count $\mathbf{Y}_e$ (GFlop)	Operation Count $\mathbf{F}_r^u$ (GFlop)	Total Operation Count (GFlop)
q500×500	1.845	9.632	11.476
f500×500	2.885	13.495	16.380
s30×30×30	2.968	12.067	15.035
f30×30×30	4.840	23.534	28.375

Table 5.4: For example test problems, operation counts for the forward elimination with 100 RHS vectors

As it is stated in the previous Chapter, the mapping found for the factorization is also used for the triangular solution. This implicitly assumes that the workload for triangular solution is directly proportional to the workload for the partial factorization. If the operation counts for the partial factorization and forward elimination are compared, one can see that this is not always the case. However, for large tree nodes with similar *ne*/*nr* ratios, the triangular solution flop is almost directly proportional to the partial factorization flops. This point is illustrated by plotting the triangular solution flops in terms of the partial factorization flops. Figure 5.3 shows the flop for the forward elimination with 100 RHS vectors normalized according to the partial factorization flop. The relative flop is given for different *ne*/*nr* ratios in Figure 5.3. As shown in Figure 5.3, the relative flops are different for two frontal matrices with significantly different *ne*/*nr* ratios. However, if the *ne*/*nr* ratios are the same, the relative flop becomes very similar for two frontal matrices as long as the frontal matrix sizes differs in only by a small constant.

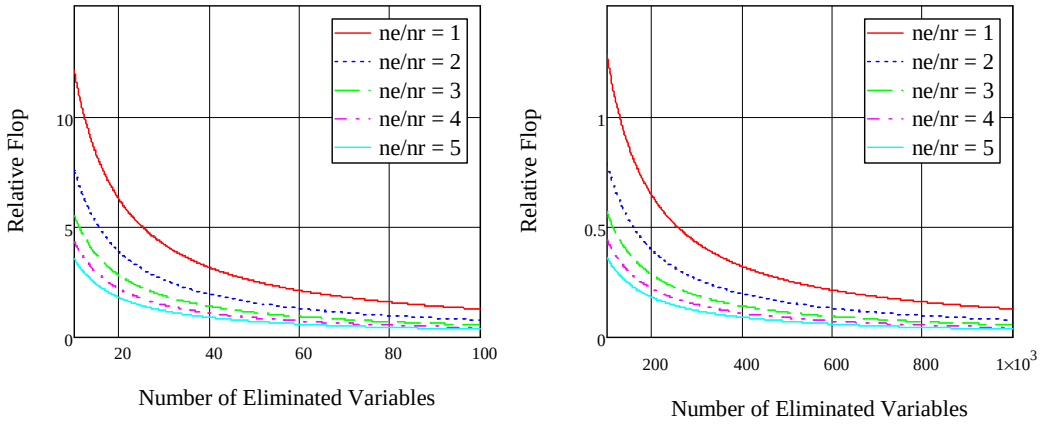


Figure 5.3: The flop required for the triangular solution with 100 RHS vectors given in terms of the flop required for the partial factorization.

After demonstrating the assumptions of the mapping algorithm for the triangular solution, we now discuss the implementation of the forward elimination. As it is shown in Figure 5.1 for an example assembly tree, the dependency between the forward elimination tasks is the same as the one for the factorization tasks. Therefore, the postorder tree traversal used in the numerical factorization can also be used for the forward elimination. The use of postorder traversal for forward elimination allows storing the $\mathbf{F}_r^u$ in an update matrix stack data structure that is similar to the one used in the factorization phase. The pseudo code for the multithreaded forward elimination is shown in Figure 5.4. First, $\mathbf{F}_e$ in the current RHS frontal matrix is initialized according to the loads on the structure. Then, the RHS frontal matrix blocks, $\mathbf{F}_e$ and $\mathbf{F}_r$, are updated according to the $\mathbf{F}_r^u$ computed at the children tree nodes. Once, the RHS frontal matrix is constructed and the factors for the current assembly tree node are restored, the forward elimination is performed by making the BLAS3 calls described previously.

The multithreaded forward elimination algorithm

INPUT:

current_thread_id: the thread id of the current thread
factors: factors calculated in the numerical factorization phase
tree_nodes: the assembly tree nodes assigned to the current thread (postorder traversal order)
$\mathbf{F}_e$, $\mathbf{F}_r$: the RHS frontal matrix for the current thread
rhs_update_stacks: the update stacks for the RHS vectors for all threads

OUTPUT:

y: the results found by forward elimination

```
/* traverse all nodes assigned to the thread */
for each tree_node in tree_nodes
  /*Wait until all children nodes that the tree node depends are processed */
  if requires_wait(tree_node) = true
     wait_for_threads(tree_node)
  end if
  assemble(loads_at_eliminated_dofs, F_e)
  for each child_node of the tree_node
     /*Get the thread id that processed the children node */
     thread_id:=get_thread(child_node)
     /*Pop the update matrix from the corresponding update stack */
     if thread_id = current_thread_id
        F_r^u:= pop(rhs_update_stacks[thread_id])
     else
        /*find the child node's update matrix in the update matrix stack*/
        F_r^u:=find(child_node, rhs_update_stacks[thread_id])
     end if
     assemble(F_r^u, F_e)
     assemble(F_r^u, F_r)
  end for
  /*Get the stored factors for the current frontal matrix*/
  L_B:=restore_diagonal(tree_node, factors)
  L_off:= restore_off_diagonal(tree_node, factors)
  /*If it is a high-level tree node then use all threads available for partial factorization */
  if is_subtree_node(tree_node) then set_blas_lapack_thread_num(1)
  otherwise set_blas_lapack_thread_num(max_thread_num) end if
  /*Call the BLAS3 functions for the forward elimination */
  F_e:=dtrsm(L_B, F_e)
  F_r^u:=dgemm(F_r, L_off, F_e)
  /*Store the results from the forward elimination*/
  push(F_r^u, load_update_stacks[current_thread_id])
  add F_e to the y
  /*Notify the threads that waits for this tree node */
  if requires_notification(tree_node) = true
     notify_threads(tree_node)
  end if
end for
```

Figure 5.4: The pseudo code for multithreaded forward elimination algorithm.

As described in Chapter 4.3, node blocking improves the performance of the assembly of the update matrices for the multifrontal factorization. We can also exploit the node blocks for the triangular solution of a frontal matrix. This will improve the efficiency of the assembly of the $\mathbf{F}_r^u$ matrices corresponding to children tree nodes. The storage scheme for the RHS vectors determines the size of the continuous matrix blocks that can be assembled at once. Consequently, the benefits from the node blocking depend on how the RHS vectors are stored. To illustrate this point, Figure 5.5 shows two alternative schemes for storing the RHS vectors. Here, it is assumed that the RHS matrix is stored in a column oriented fashion which is the case for all dense matrices used in the SES solver package. The storage scheme shown in Figure 5.5(a) is for the forward elimination formulation on a frontal matrix given in Equation 5.8. For the storage scheme shown in Figure 5.5(a), the loads of a RHS vector are adjacent in the array storing the RHS matrix. In other words, the loads on a dof corresponding to adjacent RHS vectors are separated by *ne+nr* elements in the array storing the RHS matrix. This limits the use of node blocking to a single RHS vector. For the triangular solution of multiple RHS vectors, there is a more efficient storage scheme, which is given in Figure 5.5(b). Here, the transpose of the right hand-side vectors are stored. In this scheme, all loads corresponding to a dof are adjacent. The assembly of the $\mathbf{F}_r^u$ blocks can be performed efficiently for all RHS vectors at once. This storage scheme allows the efficient use of BLAS1 kernels for the assembly of the $\mathbf{F}_r^u$'s.

In order to use the storage scheme shown in Figure 5.5(b), the forward elimination formulations given in Equations 5.9 and 5.10 must be modified as follows:

$$\mathbf{Y}_\mathbf{e}^\mathrm{T} = \mathbf{F}_\mathrm{e}^\mathrm{T}\mathbf{L}_\mathrm{B}^{-\mathrm{T}} \tag{5.14}$$

$$\mathbf{F}_\mathrm{r}^{\mathrm{u}\mathrm{T}} = \mathbf{F}_\mathrm{r}{}^\mathrm{T} - \mathbf{Y}_\mathrm{e}{}^\mathrm{T}\mathbf{L}_\mathrm{off}{}^\mathrm{T} \tag{5.15}$$

The BLAS3 functions are the same for the modified equations given above. However, the parameters of the BLAS3 functions should be changed. Now, the transpose of the factors are used in the corresponding BLAS3 functions.

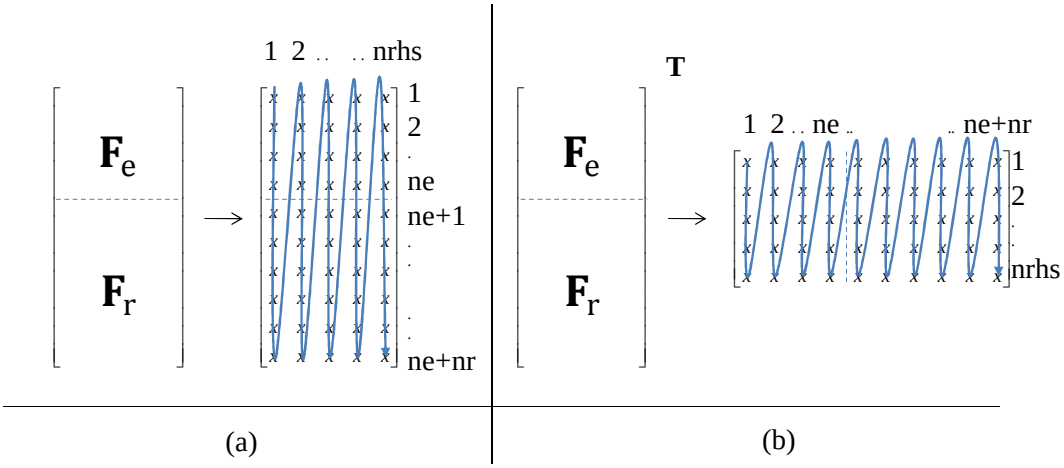


Figure 5.5: Alternative storage schemes for the RHS vectors.

5.2.2 Back Substitution

After forward elimination is finished, the solution is completed by a back substitution. For the back substitution steps, the frontal matrix and RHS vectors can be written in the blocked matrix form as follows:

$$\begin{bmatrix} \mathbf{L}_B^T & \mathbf{L}_{off}^T \\ 0 & \mathbf{X} \end{bmatrix} \begin{bmatrix} \mathbf{D}_e \\ \mathbf{D}_r \end{bmatrix} = \begin{bmatrix} \mathbf{Y}_e \\ \mathbf{Y}_r \end{bmatrix} \tag{5.16}$$

where $\mathbf{D}_e$ and $\mathbf{D}_r$ are displacements corresponding to the eliminated and remaining dofs. $\mathbf{D}_e$ is *ne* by *nrhs* matrix and $\mathbf{D}_r$ is *nr* by *nrhs* matrix. The back substitution for the current frontal matrix determines $\mathbf{D}_e$. The entries of $\mathbf{Y}_e$ are the values computed in the forward elimination phase from Equation 5.9. $\mathbf{D}_r$ stores the displacements for the remaining dofs, which are found at an ancestor of the current tree node. $\mathbf{X}$ and $\mathbf{Y_r}$ are not used for the back substitution operations on the frontal matrix. Matrix operations for the back substitution are given as:

$$\mathbf{Y}_e^u = \mathbf{Y}_e - \mathbf{L}_{off}^T \mathbf{D}_r \tag{5.17}$$

$$\mathbf{D}_e = \mathbf{L}_B^{-T}\mathbf{Y}_e^u \tag{5.18}$$

The BLAS3 dgemm and dtrsm functions are used to perform operations given in Equations 5.17 and 5.18 respectively. After the displacement matrix $\mathbf{D}_e$ is computed, it is disassembled to $\mathbf{D}_r$ matrices of the children elements ($\mathbf{D}_r^c$). The number of floating point operations required for the back substitution is the same as the ones for the forward elimination, which are given in Equation 5.11 and Equation 5.12.

As shown in Figure 5.1, the back substitution starts from the parent node in the assembly tree and then continues in the reverse direction of the factorization and forward elimination. Therefore, instead of the postorder tree traversal used in the factorization and forward elimination, the reverse of the postorder tree traversal is used in the back substitution. The pseudo code for the multithreaded back substitution is given in Figure 5.7. Here, the RHS update matrix stack that stores $\mathbf{F}_r^u$ for the forward elimination can be used to store $\mathbf{D}_r^c$ matrices where $\mathbf{D}_r^c$ matrices are the $\mathbf{D}_r$ matrices used in the back substitution steps at the children tree nodes. $\mathbf{D}_r^c$ matrices are placed on the stack in the reverse order of the placements in the forward elimination steps. As shown in Figure 5.7, before the back substitution operations, the results found in forward elimination are restored. Then, the $\mathbf{D}_r^c$ is popped from the RHS update matrix stack and it is restored at the RHS frontal matrix $\mathbf{D}_r$. Back substitution operations for the assembly tree are performed by using the matrices $\mathbf{D}_e$, $\mathbf{D}_r$, $\mathbf{L}_B$, and $\mathbf{L}_{off}$. After the back substitution operations are completed, the RHS frontal matrix blocks $\mathbf{D}_e$ and $\mathbf{D}_r$ are assembled to the $\mathbf{D}_r^c$. $\mathbf{D}_r^c$ will be used during the back substitution operations on the children tree nodes.

Similar to the storage scheme in the forward elimination, the transpose of the RHS frontal matrix is stored for the back substitution in order to exploit the node blocks for multiple RHS vectors. The formulation for the transpose RHS frontal matrices are given as follows:

$$\mathbf{Y}_e^{uT} = \mathbf{Y}_e{}^T - \mathbf{D}_r{}^T\mathbf{L}_{off} \tag{5.19}$$

$$\mathbf{D}_e^{\,T} = \mathbf{Y}_e^{u\,T}\mathbf{L}_B^{-1} \tag{5.20}$$

The BLAS3 functions are the same for the transposed equations. However, the function parameters are different than the ones for the Equations 5.19 and 5.20.

The triangular solution algorithm developed in this study is especially efficient for the solution with a large number of RHS vectors. The main advantage of this scheme is that the BLAS3 functions are performed on large dense matrices. The use of update matrix stacks for the RHS vectors is the overhead for performing operations on dense frontal matrices. For systems with a large number of RHS vectors, the increased performance of BLAS3 kernels improves the overall performance for the triangular solution regardless of the extra operations required for handling the update matrices. This point is illustrated by comparing the solution time of the proposed scheme with the solution time of the PARDISO solver. Figure 5.6 gives the triangular solution time of the SES solver package in terms of the solution time of the PARDISO solver for the test problem f500×500. As shown in Figure 5.6, PARDISO is faster for the triangular solution with 10 RHS vectors. However, as the number of RHS vectors increases, the performance of SES relative to PARDISO increases.

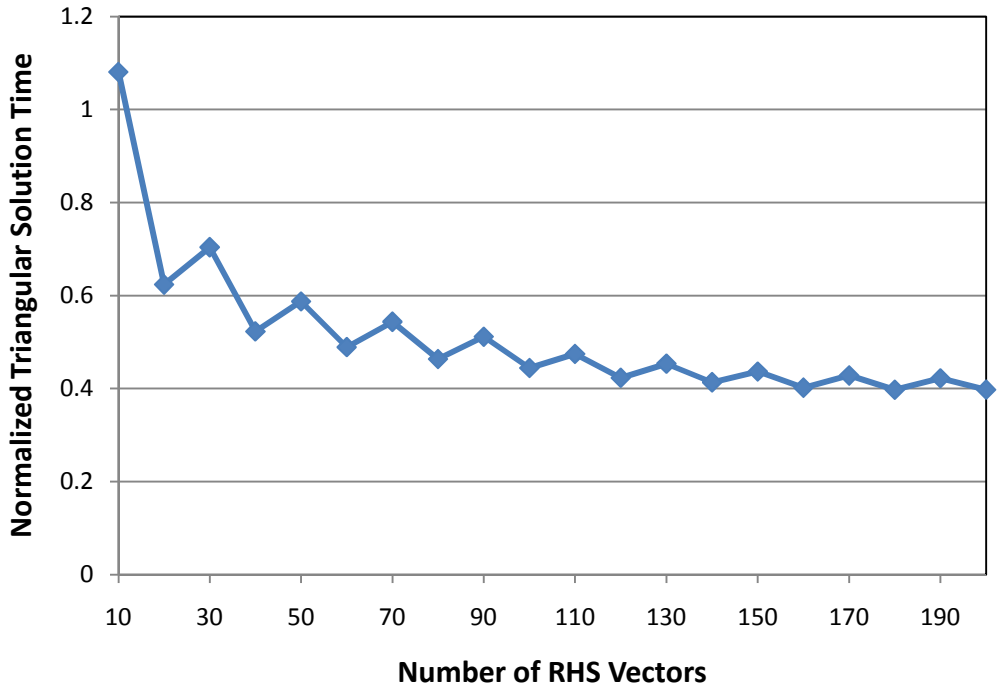


Figure 5.6: SES triangular solution time relative to the PARDISO triangular solution time for the problem f500×500.

The multithreaded back substitution algorithm

INPUT:

current_thread_id: the thread id of the current thread
factors: factors calculated in the numerical factorization phase
y: the results found in the forward elimination
tree_nodes: the assembly tree nodes (in reverse postorder traversal order)
$\mathbf{D}_e$, $\mathbf{D}_r$: the RHS frontal matrix for the current thread
rhs_update_stacks: the update stacks for the RHS vectors for all threads

OUTPUT:

d: the unknowns(displacements) found by back substitution

```
/* Traverse all nodes assigned to the thread */
for each tree_node in tree_nodes
    /*Wait until all children nodes that the tree node depends are processed */
    if requires_wait(tree_node) = true
       wait_for_notification(tree_node)
    end if
    /*Get the stored factors*/
    L_B:=restore_diagonal_factors(factors)
    L_off:= restore_off_diagonal_factors(factors)
    /*Get the forward elimination results for the current tree node*/
    D_e:=get(tree_node, y)
    /*Pop the calculated unknowns stored by the parent*/
    D_r:=pop(rhs_update_stacks[thread_id])
    /*If it is a high-level tree node then use all threads available for partial factorization */
    if is_subtree_node(tree_node) then set_blas_lapack_thread_num(1)
    otherwise set_blas_lapack_thread_num(max_thread_num) end if
    D_e:=dgemm(D_e, L_off, D_r)
    D_e:=dtrsm(L_B, D_e)
    Store D_e in d
    for each child_node of the tree_node
        /*Dissemble D_e for the child node*/
        D_r^c:=dissemble(child_node, D_e)
        /*Dissemble D_r for the child node*/
        D_r^c:=dissemble(child_node, D_r)
        /*Get the thread id that will process the children node*/
        thread_id:=get_thread(child_node)
        /*Push the calculated unknowns to the update stack of the child*/
        push(D_r^c, rhs_update_stacks[thread_id])
    end for
    /*Notify the threads that waits for this tree node*/
    if requires_notification(tree_node) = true
       notify_threads(tree_node)
    end if
end for
```

Figure 5.7: The pseudo code for the multithreaded back substitution algorithm.

5.3 Using a File Storage for the Factors

For 3D problems, the memory required to store the factors increases dramatically as the number of elements used in three dimensions increases. Figure 5.8 shows the memory required for storing the factors for problems with cubic geometry modelled with 8-node solid elements. The x-axis shows the number of elements used for each dimension of the cube. In Figure 5.8, the memory required to store the factors shows a cubic growth as a function of the number of elements on one side of the cube. For example, the memory size required to store the factors is about 10 Gbytes for a cubic model with 60 elements on each side, whereas, it is 25 Gbytes for the model with 75 elements on each side. Figure 5.8 also shows the storage requirements if the factors are not stored in the main memory. The active memory is the memory required for holding the frontal matrix and update matrix stacks. As shown in Figure 5.8, the active memory requirement can be significantly smaller than the memory required to store the factors. In a naive implementation of the multifrontal method, both active memory and the factors will be stored in the main memory. Therefore, the total memory required for storing the floating point numbers is the summation of the two plots given in Figure 5.8. The memory requirements of the space frame models are even larger since there are 6 dofs at each node for space frames.

In the multifrontal method, the factors can be written to disk after the partial factorization is completed for a frontal matrix. This can significantly reduce the memory requirements for the numerical solution. The factors will be read from the disk when they are needed again during the triangular solution. If we employ a secondary storage for the factors, the maximum memory required by the solver is a function of the maximum size of the frontal matrix and update matrix stack, which may be significantly smaller than the memory required for the factors as shown in Figure 5.8.

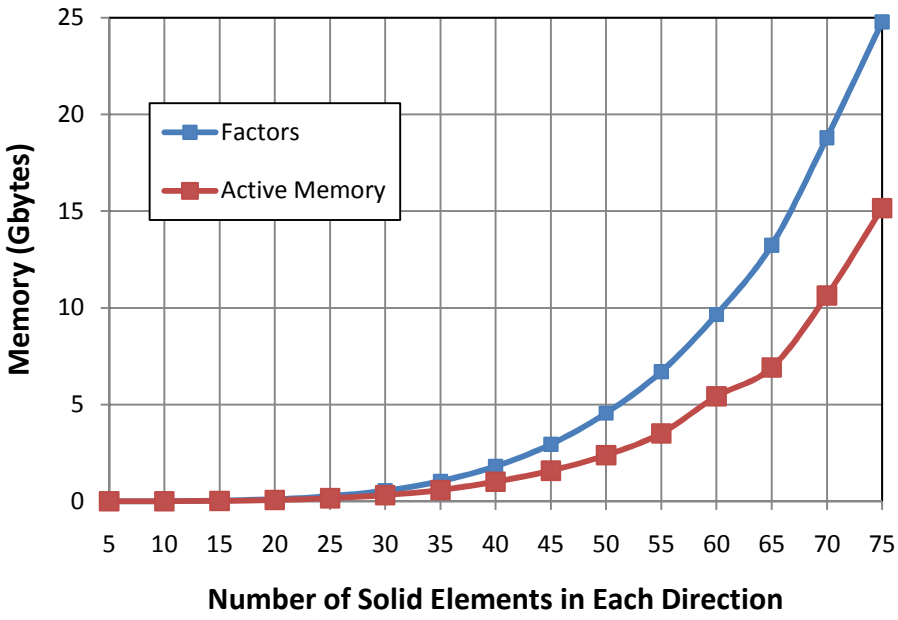


Figure 5.8: The memory requirements for the factorization of cubic geometry 8 node solid element models. HMETIS is used for the pivot-ordering.

In order to reduce the memory requirements of the multifrontal solver, an out-of-core multifrontal method is implemented by writing the computed factors to a file. This is a partial out-of-core implementation since the frontal matrix and update matrices are always kept in the main memory. For a fully out-of-core solver implementation, the frontal matrix and update matrices can also be written to a secondary storage when there is not enough memory to store them.

Acharya et al. [150] discussed tuning I/O intensive parallel applications. They demonstrated the efficiency of one coalesced large I/O request instead of a number of small I/O requests. They stated that with code restructuring, small I/O requests can be converted to larger coalesced I/O requests. According to their study, for the large I/O requests, the effect of I/O interface on performance was not as significant as it is for the small I/O requests. They also reported that prefetching data and writing-behind can improve the performance of I/O intensive applications. We considered these optimizations for writing and reading the computed factors.

For writing factors to a file, we consider two alternative approaches: synchronous and asynchronous I/O. In the synchronous I/O, a thread enters to a wait state until the I/O request is completed. In asynchronous I/O, on the other hand, the thread continues to execute after a successful I/O request. Once an asynchronous I/O request is served by the system, the thread is signalled. The signal state can be queried anytime by the thread to check whether an I/O request is completed. If the I/O request is completed, the operations on the requested data are performed.

In order to guarantee a truly asynchronous behaviour on Windows systems, the I/O operations should be explicitly buffered [151]. Therefore, we used a factor buffer to store the computed factors before writing them to a file. Figure 5.9 shows the data structures used to write the factors to the disk. As shown in Figure 5.9, the computed factors are first copied to the factor buffer. When the buffer size exceeds a certain limit, it is flushed to a file on the disk. By copying the factors to a buffer first, we reduce the number of file accesses and increase the size of the data written to the disk at each I/O request. These are expected to improve the performance of I/O intensive applications as described by Acharya et al. [150]. There are some caveats for using an explicit file buffer on Windows systems [151]. First, the number of bytes accessed should be a multiple of the volume sector size. Second, buffer addresses shall be aligned on addresses in memory that is a multiple of the volume sector size. Our buffer implementation for the factors satisfies these requirements.

In order to better exploit the asynchronous I/O, we used two factor buffers. A buffer is used as a disk cache for holding the factors that are currently written to a file. Meanwhile, the other buffer is used as a temporary storage for the factors that are recently calculated or will be calculated soon. Once the buffer caching for the disk is completely flushed to a file, the two buffers are swapped. After swapping the buffers, the buffer that was holding the most recent factors serves as a disk cache and vice versa.

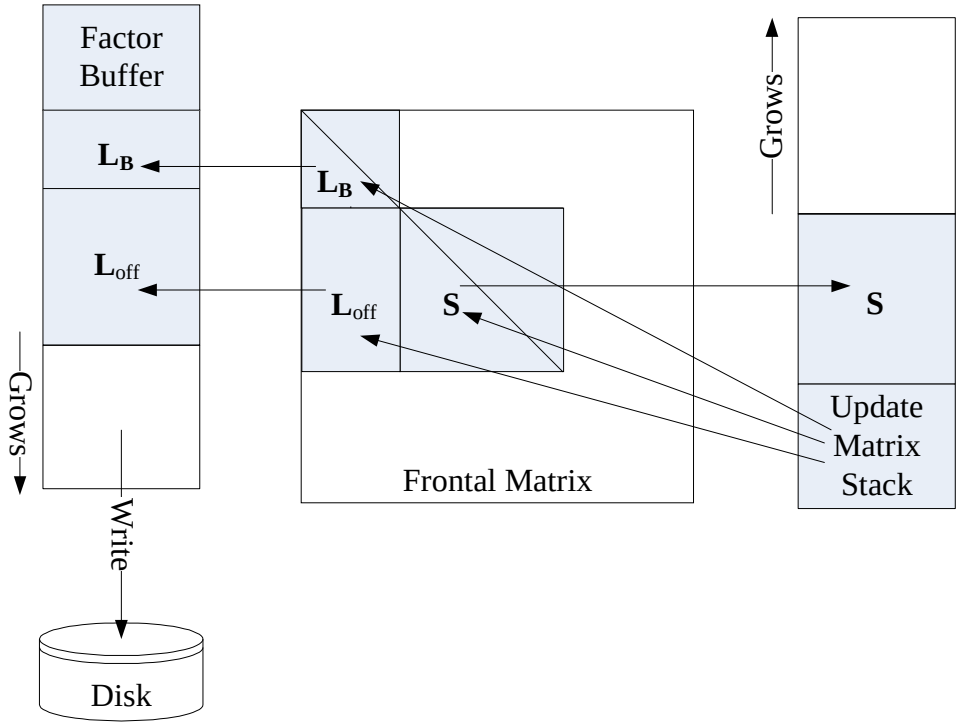


Figure 5.9: Data accesses for partial factorization on a frontal matrix

For the triangular solution, the factors need to be read from the file. We can use either synchronous or asynchronous I/O for reading the factors from the file. Similar to writing factors to a file, an explicit file buffer is required for guaranteeing truly asynchronous reading. Figure 5.10 shows the data structures used for reading the factors from a file. As shown in Figure 5.10, we use a factor buffer to read the factors for the triangular solution. In addition, since we know what factors will be needed at the next step, we can fetch the data before it is required. Compared to the I/O requests right before the operations on the required data, a prefetching scheme can potentially reduce the waiting time for vital data, which are the factors computed in the numerical factorization phase. We again used two buffers to read the factors. The factors are read in large blocks by using the double buffer structure described before and used for storing the factors.

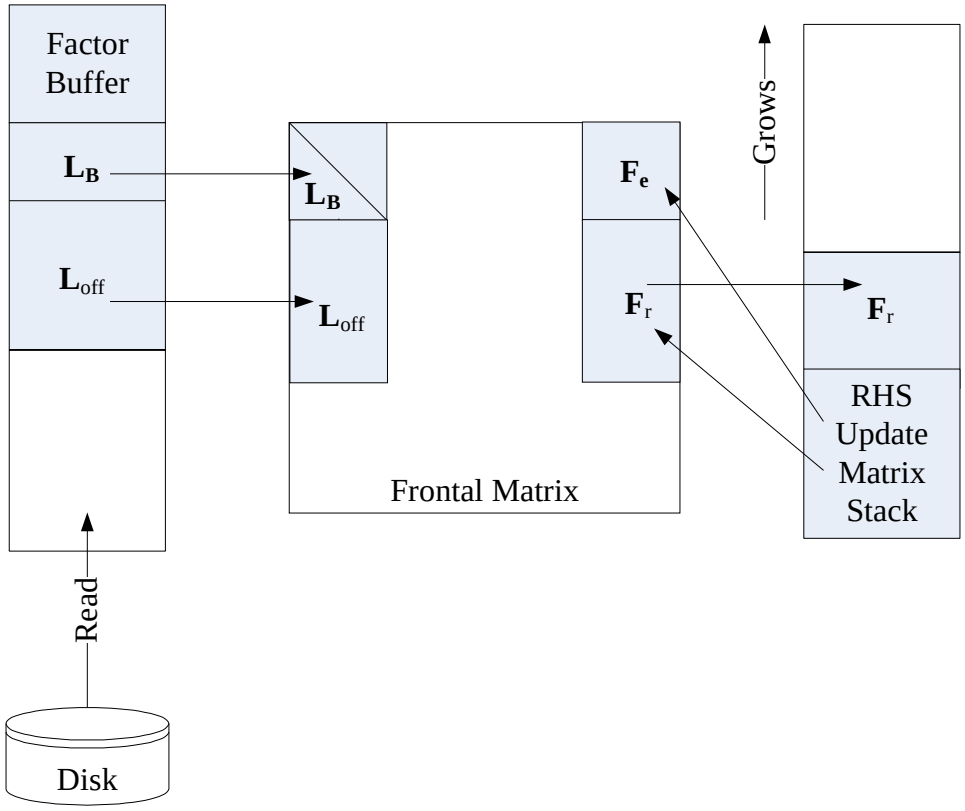


Figure 5.10: Data accesses for partial forward elimination on a frontal matrix

The number of disk reads will be halved if the forward elimination for a frontal matrix is performed right after the partial factorization of the frontal matrix. In this case, the factors are read from the file only once during the back substitution. We employ this scheme in order to minimize the disk writes. The disk writes can be further reduced by creating a factor buffer as large as the available memory size and not writing the factors to a file unless the memory required to store factors exceeds the available memory. However, this scheme is not implemented yet.

For the multifrontal solver, the memory required for the update matrix stack can be reduced by employing the scheme described by Guermouche and L'excellent [55]. This will further reduce the active memory required for the multifrontal solver and increase the size of the problems that can be solved using the out-of-core solver. However, this scheme is not implemented in the solver package yet.

CHAPTER 6

PERFORMANCE OF VARIOUS ALGORITHMS

In this chapter, numerical experiments are performed to evaluate the performance of alternative algorithms for a large number of 2D and 3D FE problems. The algorithms that produce favorable execution times are presented. Approaches to improve the overall efficiency of the sparse direct solution are discussed.

6.1 Performance of Matrix Ordering Programs

6.1.1 Program Parameters

Matrix-ordering programs have adjustable parameters and the SES solver package allows experimenting with these parameters as described in Section 3.3. Here, we investigate the performance of using different values for some of these parameters for the matrix ordering programs.

6.1.1.1 Graph Compression

A graph compression reduces the size of the input graph and typically reduces the execution time of matrix-ordering programs [129, 136]. Here, we investigate the effect of graph compression for the HMETIS matrix ordering program. Figure 6.1 shows the performance profiles for non-zero with graph compression and no graph compression. The benchmark suite of 40 test problems is used for the numerical experiments. As shown in Figure 6.1, graph compression usually yields pivot-orderings with slightly more non-zeros. For a test problem, compressing the input graph yields a non-zero value 1.14 times the non-zero without compression. Figure 6.2 shows the impact of graph compression on flop for HMETIS matrix ordering program. As shown in Figure 6.2, graph compression usually yields higher flop values for HMETIS. Compared to the non-

zero, the difference between the flop values is more significant. For a single problem in our benchmark suite, graph compression yields flop values 1.5 times the flop found without graph compression.

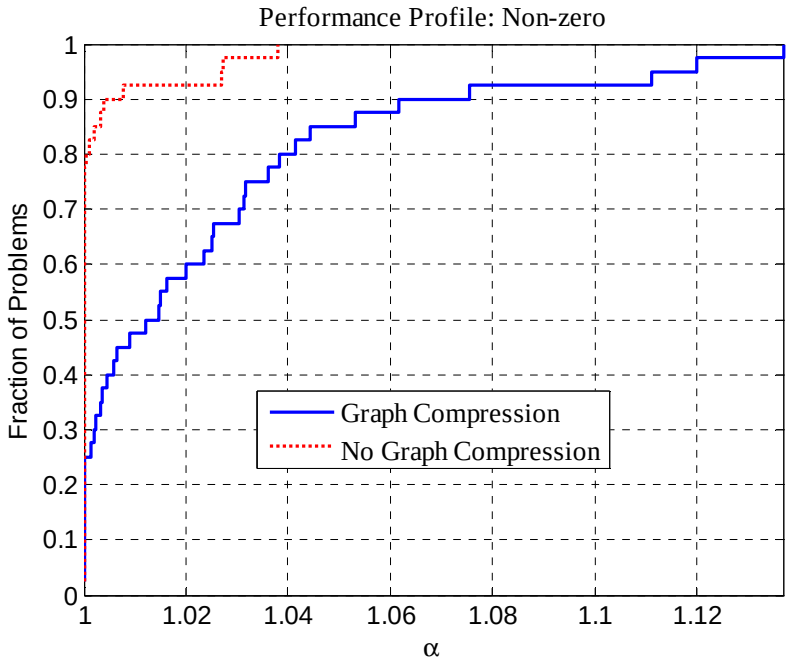


Figure 6.1: Performance profile, $p(\alpha)$: Non-zero for HMETIS with and without graph compression, benchmark suite of 40 test problems

Although the graph compression usually increases the non-zero and flop for the pivot-orderings, the execution time of the HMETIS is smaller if the graph compression is employed. For the benchmark suite of 40 test problems, the matrix-ordering may be 1.7× slower if the graph is not applied. The reduction in matrix ordering time may be important since time spent in the matrix-ordering time may correspond to a significant portion of the solver execution time, especially for 2D problems. However, for 2D problems, local orderings typically yields better pivot-orderings with respect to non-zero and flop. For 3D problems, the matrix ordering times may be significantly smaller than the factorization times especially for large problems. Therefore, it may be desirable to reduce the factorization times at a relatively small cost of matrix ordering times for 3D

problems. For the hybrid ordering program HAMF, the graph compression has minor impact both on the matrix-ordering times and the quality of the pivot-orderings.

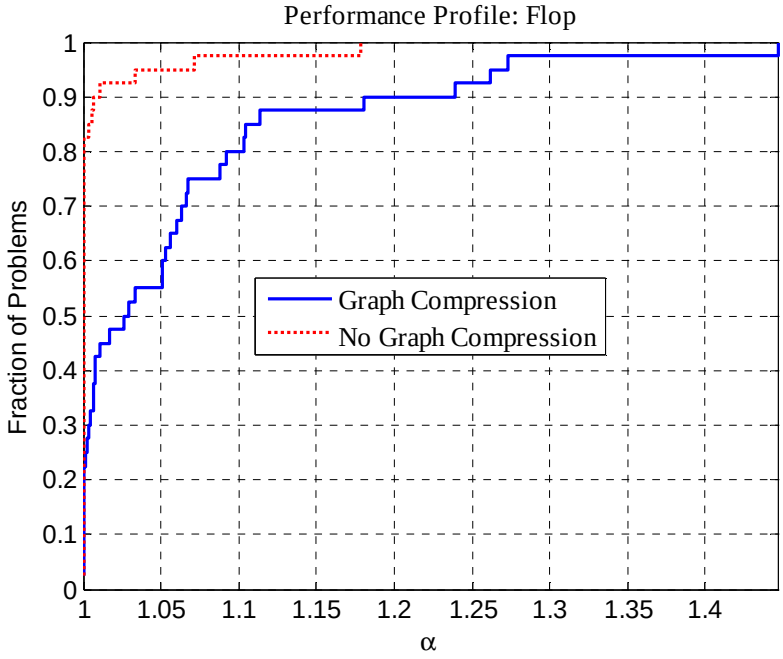


Figure 6.2: Performance profile, $p(\alpha)$: Flop for HMETIS with and without graph compression, benchmark suite of 40 test problems

6.1.1.2 Nested Dissections Stopping Criteria, *vertnum*, for HAMF

In the hybrid matrix ordering programs, HMETIS and HAMF, the nested dissections are performed until the partitions are smaller than a threshold value. Once the partitions are found with incomplete nested dissections, the partitioned graphs are ordered using a local matrix ordering program. The stopping criterion, *vertnum*, for the nested dissections is an input for the HAMF matrix ordering program. For partitions smaller than the *vertnum* value, the nested dissections are stopped and a local matrix-ordering is used for the partitions. We investigate the performance of HAMF for alternative values of *vertnum*. Figure 6.3 shows the flop for alternative values of *vertnum* parameter. As shown in Figure 6.3, there is no single *vertnum* value that gives the best flop for all test problems in the benchmark suite. The performance profiles for *vertnum* values smaller than 500 are

similar as shown in Figure 6.3. The execution time of HAMF typically decreases as we increase the *vertnum* values. This is due to the fact that the time consuming graph partitioning algorithm is executed fewer times as we increase the *vertnum* values. In the limit, if *vertnum* is equal to the size of the input graph, then the local ordering algorithm is applied to the entire graph and no graph partitioning is applied. Observing that the performance for alternative *vertnum* values are similar within the range of 50 to 350, we use the default *vertnum* value in the SCOTCH library, which is *vertnum*=240.

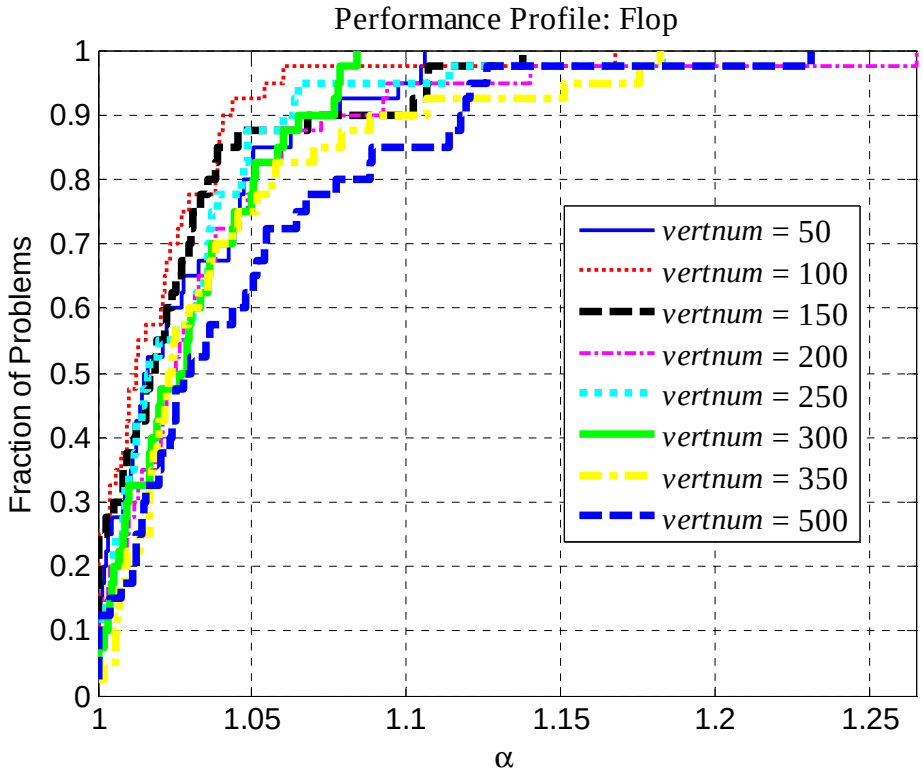


Figure 6.3: Performance profile, $p(\alpha)$: Flop for HAMF with alternative values for *vertnum*, benchmark suite of 40 test problems

6.1.1.3 Node Amalgamation within SCOTCH Library

Matrix ordering programs in SCOTCH package can find pivot-orderings with amalgamated supernodes. The *cmin* and *frat* parameters control the amount of node amalgamation within SCOTCH (see Chapter 3 for details). The numerical experiments

are performed to compare the SCOTCH amalgamation and explicit node amalgamation described in Chapter 4.2. The numerical experiments on benchmark problems show that the explicit node amalgamation typically yields fewer update operations to flop ratios compared to node amalgamation within the SCOTCH package. Consequently, the factorization time for explicit amalgamation is usually smaller than the amalgamation within the SCOTCH package.

6.1.1.4 Multiple Elimination Parameter, *delta*, in MMD

Alternative values for *delta* parameter of the MMD program are investigated for the benchmark suite of 40 test problems. Figure 6.4 shows the impact of different *delta* values on flop found with MMD. As shown in Figure 6.4, there is no single delta value that yields the best flop for all test problems. In addition, the matrix ordering times are similar for different *delta* values. For the remainder of this study, *delta* = 4 is used unless otherwise is stated.

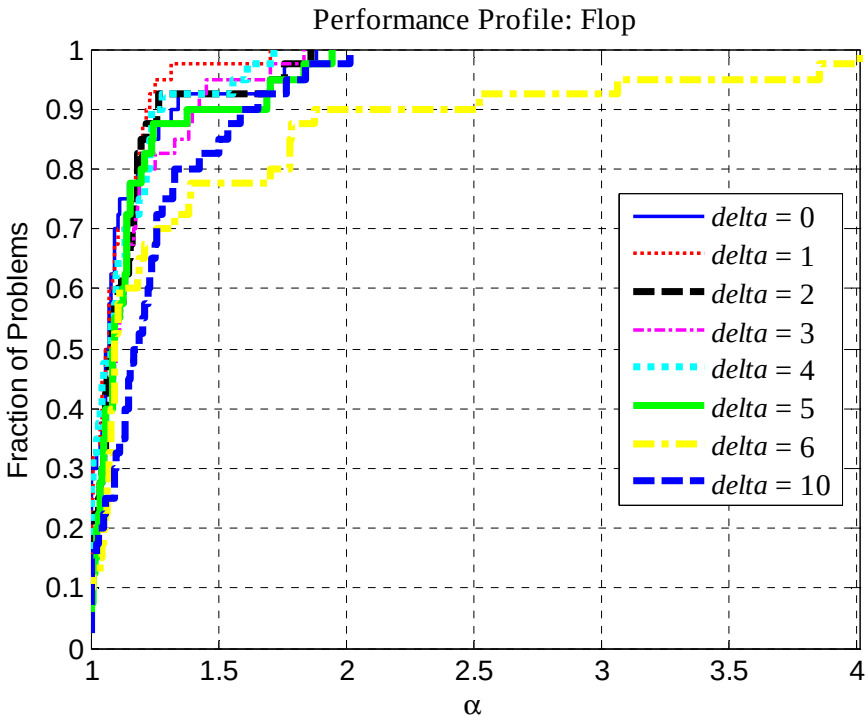


Figure 6.4: Performance profile, $p(\alpha)$: Flop for MMD with different *delta* values, benchmark suite of 40 test problems

6.1.2 Effect of Initial Node Numbering

As it is discussed in Chapter 3.2, the initial numbering of the nodes impacts the fill-ins for a pivot-ordering found by the matrix-ordering programs. First, we investigate the impact of node numberings described in Chapter 3.2 for the local orderings. Figure 6.5 shows the impact of initial node numbering for AMF ordering. As shown in Figure 6.5, coordinate based ordering for the initial node numbering yields to the most favorable flop values for approximately 90% of the test problems with regular geometries. Figure 6.6 shows the impact of initial node numberings for MMD ordering. Similar to the results for AMF, coordinate based initial node numbering produces the minimum flop values for the majority of the test problems with regular geometries. For the CAMD ordering, on the other hand, the improvements in flop values due to the coordinate based initial numbering are not as significant as it is for the other two local ordering programs. Figure 6.7 shows

the impact of initial node numberings for CAMD ordering. Here, the performance profiles for coordinate based ordering and reverse Cuthill-McKee ordering are similar. In addition, the gap between the coordinate based ordering and random node permutations is smaller for CAMD compared to the performance gap between them for AMF and MMD.

The performance profiles for the best pivot-ordering chosen among 15 random node permutations are also given in Figure 6.5-6.7. As shown in these figures, the use of best pivot-ordering among the alternatives for the random node permutations does not improve the performance of the local orderings significantly. The flop for a single random node permutation is comparable to the best flop chosen among 15 random node permutations. Furthermore, the coordinate based node ordering and reverse Cuthill-McKee ordering yield better performance profiles for flop compared to the one for choosing the best flop among 15 random node permutations.

Numerical experiments on the models with irregular geometries also illustrate the efficiency of the coordinate based ordering. For AMF, Figure 6.8 shows the performance profiles for the flop for the models with irregular geometries. As shown in Figure 6.8, coordinate based initial node numbering yields the most favorable flop for most of the test problems. For other local orderings AMD and MMD, the relative performance of initial node numberings for irregular geometries is similar to the ones for the regular geometries.

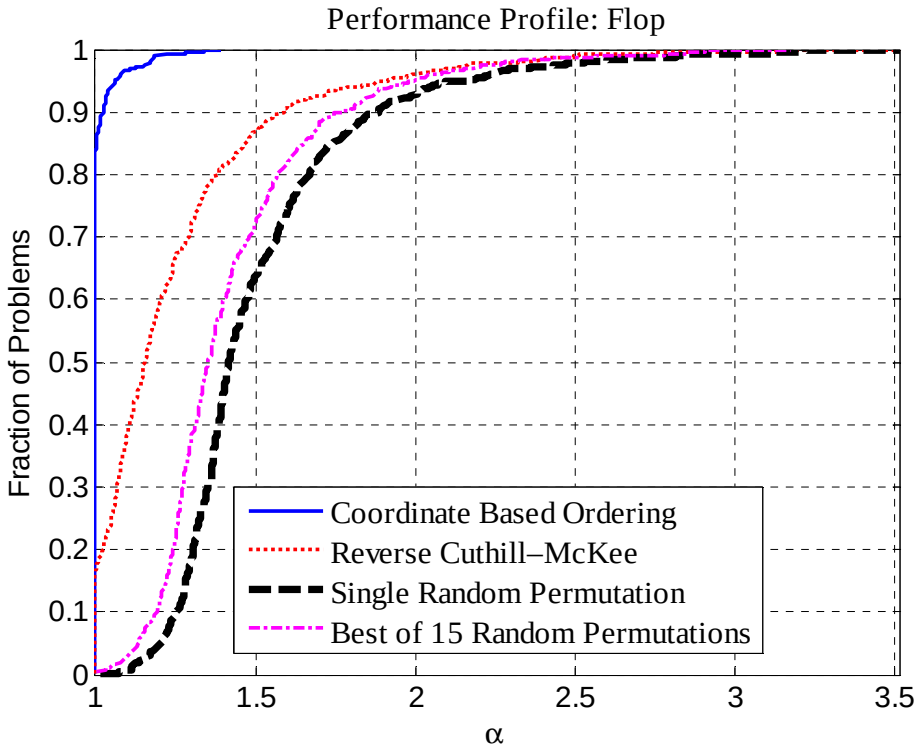


Figure 6.5: Performance profile, $p(\alpha)$: Flop for AMF with different initial node numberings, 670 test problems with regular geometries

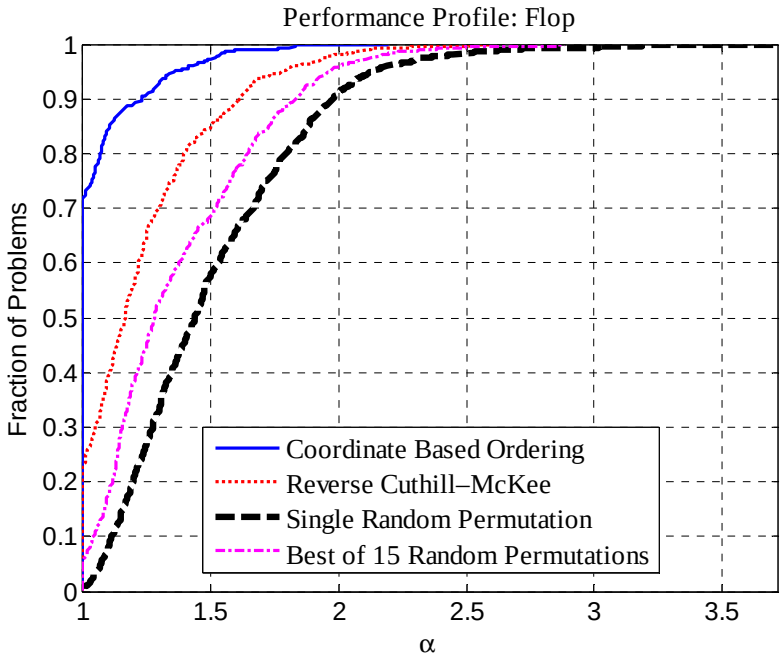


Figure 6.6: Performance profile, $p(\alpha)$: Flop for MMD with different initial node numberings, 670 test problems with regular geometries

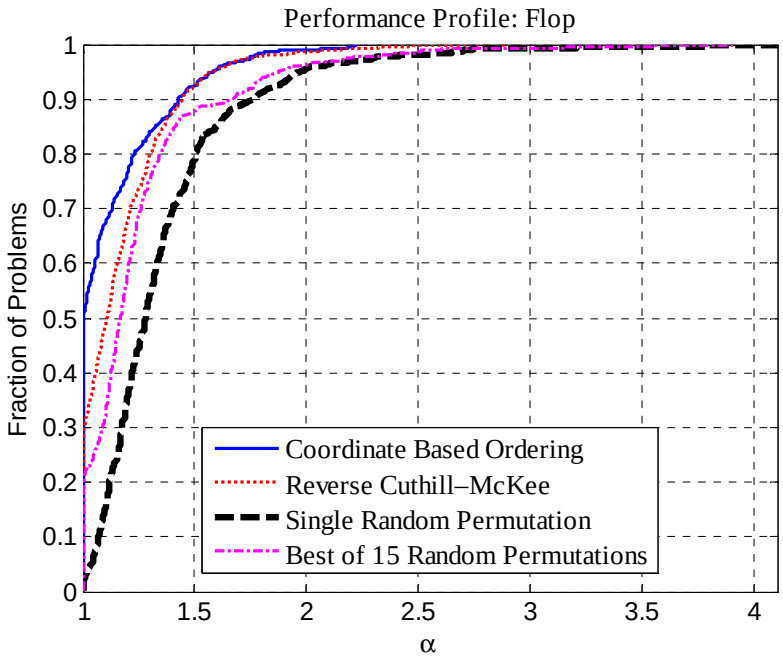


Figure 6.7: Performance profile, $p(\alpha)$: FLOP for CAMD with different initial node numberings, 670 test problems with regular geometries

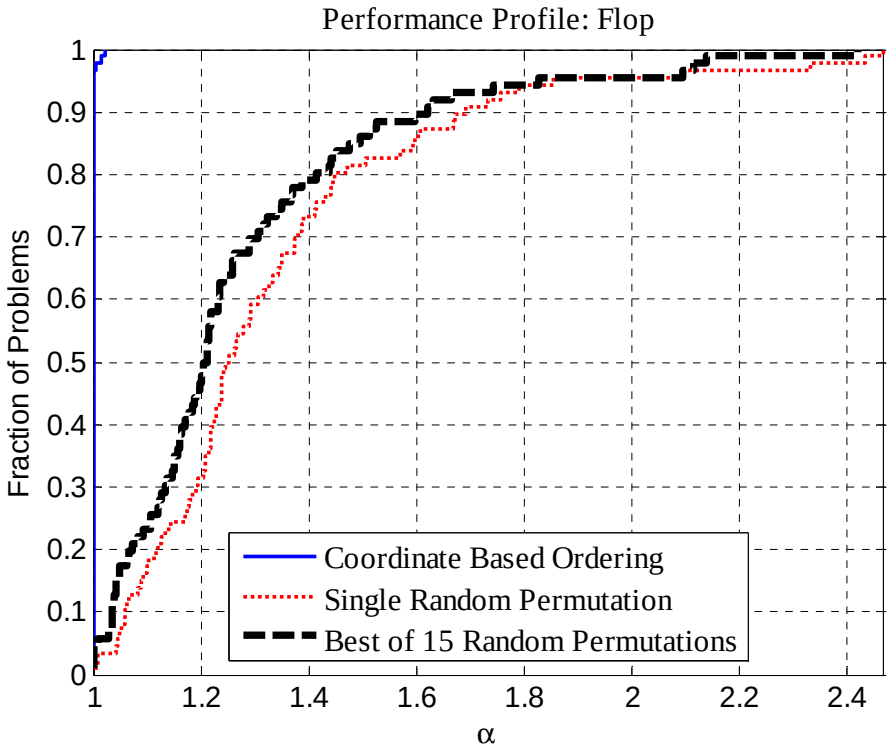


Figure 6.8: Performance profile, $p(\alpha)$: Flop for AMF with different initial node numberings, 86 test problems with irregular geometries

Next, we investigate the impact of alternative initial node numberings on pivot-orderings found with the hybrid matrix ordering programs, HMETIS and HAMF. Figure 6.9 shows the impact of initial node numberings on the flop produced by the HMETIS. Figure 6.10 shows the same information for HAMF. Unlike the local ordering programs, the coordinate based ordering offers no advantage for the HMETIS and HAMF orderings. For these hybrid ordering programs, the coordinate based orderings yield flop values comparable to the random node permutations. Therefore, it is concluded that the hybrid ordering programs are less affected by the initial node numberings and they are more robust in that sense. However, as shown in Figure 6.9 and 6.10, a smaller flop value can be obtained by executing the hybrid ordering programs for several random node permutations and using the best pivot ordering that gives the smallest flop value. Nevertheless, this approach offers moderate improvements for the flop values. As shown

in Figure 6.9 and 6.10, a random node permutation yields a flop value that is within 1.5 times the best flop value found for the random node permutations. For about 90% of the test problems with uniform geometry, a random node permutation yields flops within 1.2 times the best.

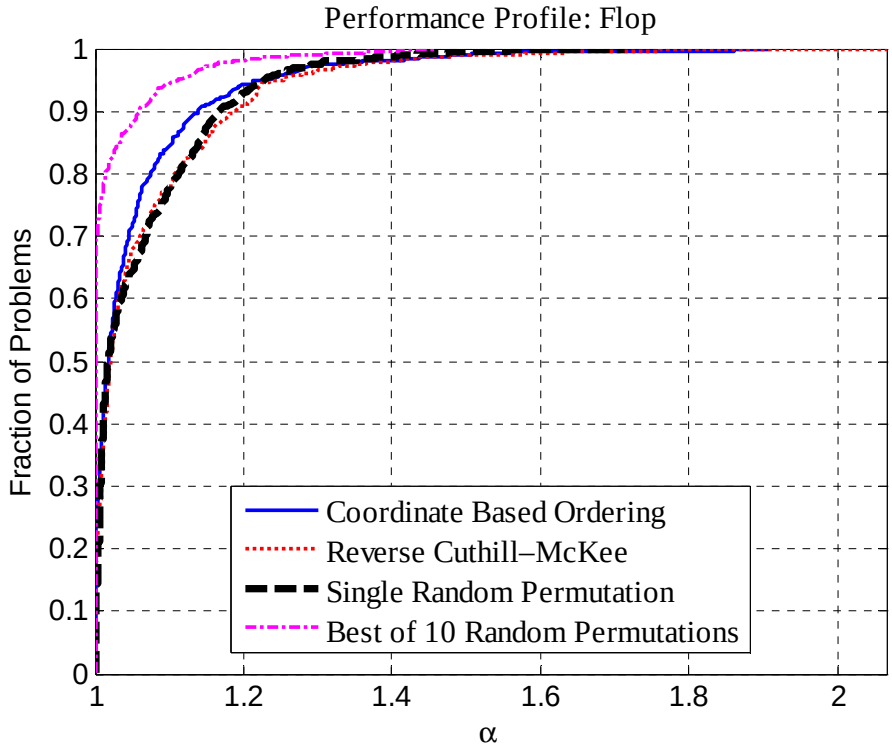


Figure 6.9: Performance profile, $p(\alpha)$: FLOP for HMETIS with different initial node numberings, 670 test problems with regular geometries

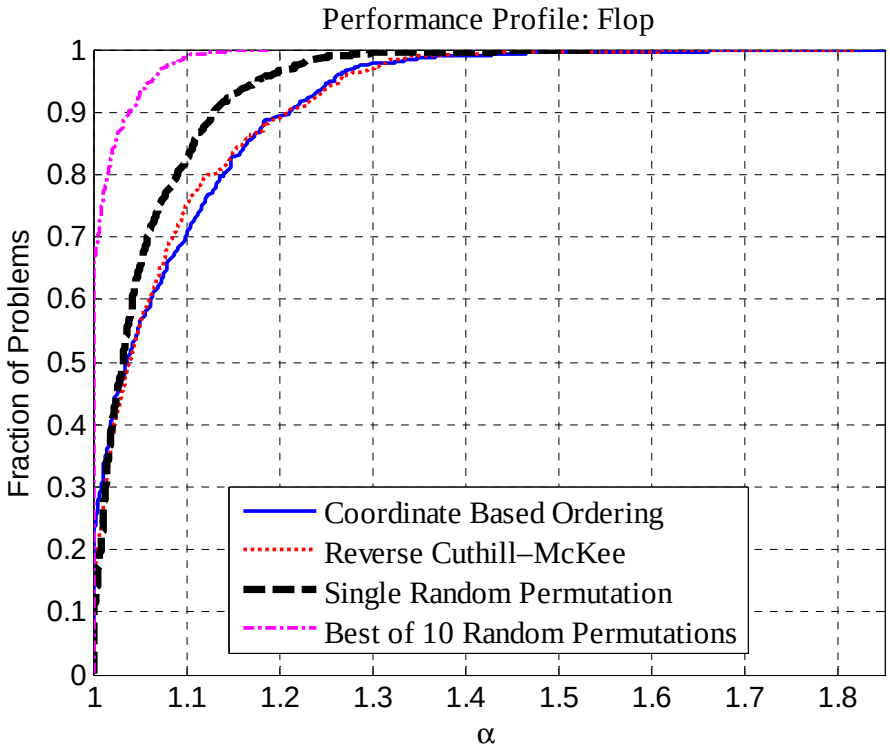


Figure 6.10: Performance profile, $p(\alpha)$: FLOP for HAMF with different initial node numberings, 670 test problems with regular geometries

6.1.3 Matrix Ordering for Serial Factorization

The serial factorization performance of alternative matrix ordering programs are evaluated. A coordinate based initial node numbering is used for the comparison of the matrix ordering programs. The relative performance of matrix ordering programs varies depending on the model dimensionality and average node connectivity of the test problems.

6.1.3.1 2D Models

For five matrix ordering programs, Figure 6.11 shows the performance profiles for non-zero. This figure shows the results for 2D models with uniform geometries. As shown in Figure 6.11, AMF gives the best non-zero for about 80% of the test problems. Furthermore, it is within 1.05 of the best non-zero among results from the remaining four

matrix ordering programs. Consequently, AMF ordering usually minimizes the factorization memory requirements for 2D models with regular geometries.

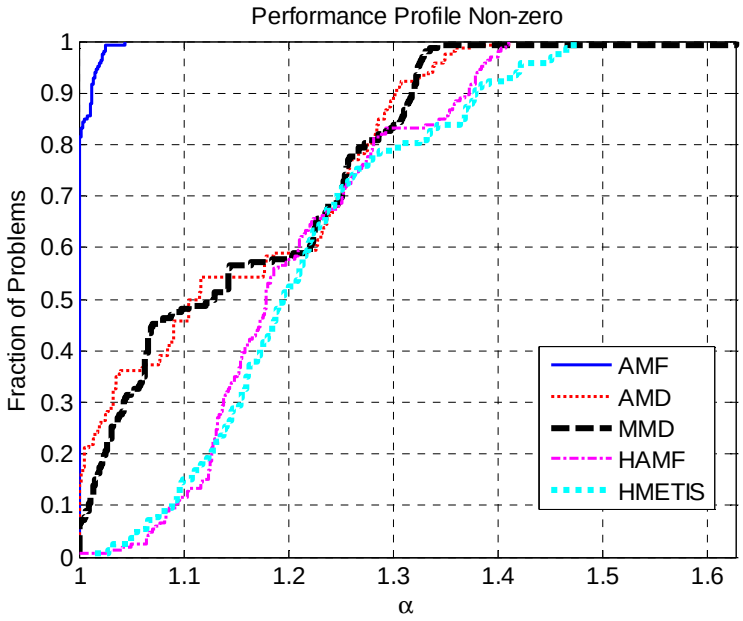


Figure 6.11: Performance profile, $p(\alpha)$: Non-zero for alternative matrix ordering programs, 166 2D test problems with regular geometries

Figure 6.11 shows the performance profiles for flop. The relative performance of matrix ordering programs for non-zero is similar to the one for the flop. As shown in Figure 6.12, AMF gives the best flop for about 80% of the test problems. Compared to the hybrid ordering programs, HAMF and HMETIS, local orderings, AMF, AMD, and MMD, usually yield fewer flop values. The flop is a measure of the factorization time. Therefore, the performance of matrix ordering programs in terms of minimization of the factorization time is expected to be similar to that for flop. The performance profiles for the PARDISO factorization times are shown in Figure 6.13. As expected, the performance profiles for factorization times are qualitatively similar to the performance profiles for the flop. The performance profiles for SES factorization times are similar to those for PARDISO.

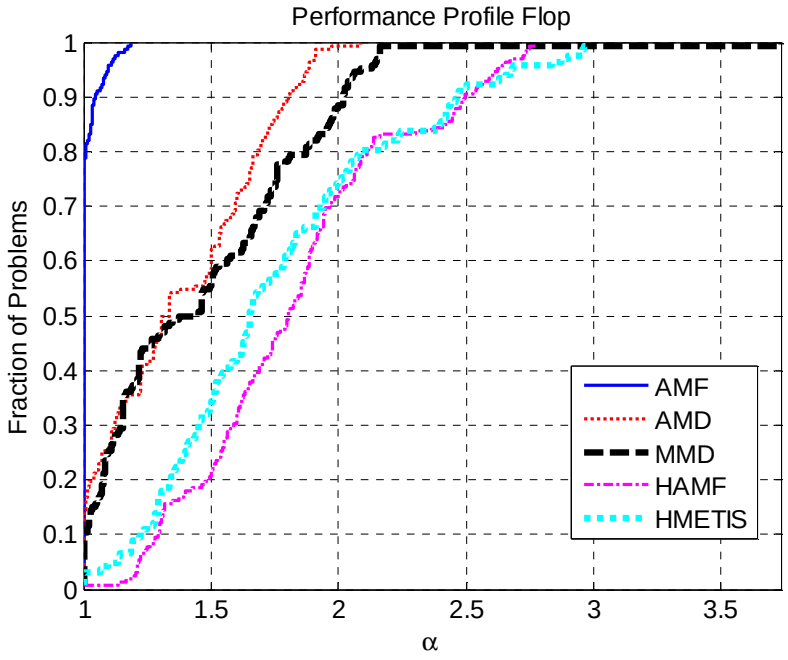


Figure 6.12: Performance profile, $p(\alpha)$: Flop for alternative matrix ordering programs, 166 2D test problems with regular geometries

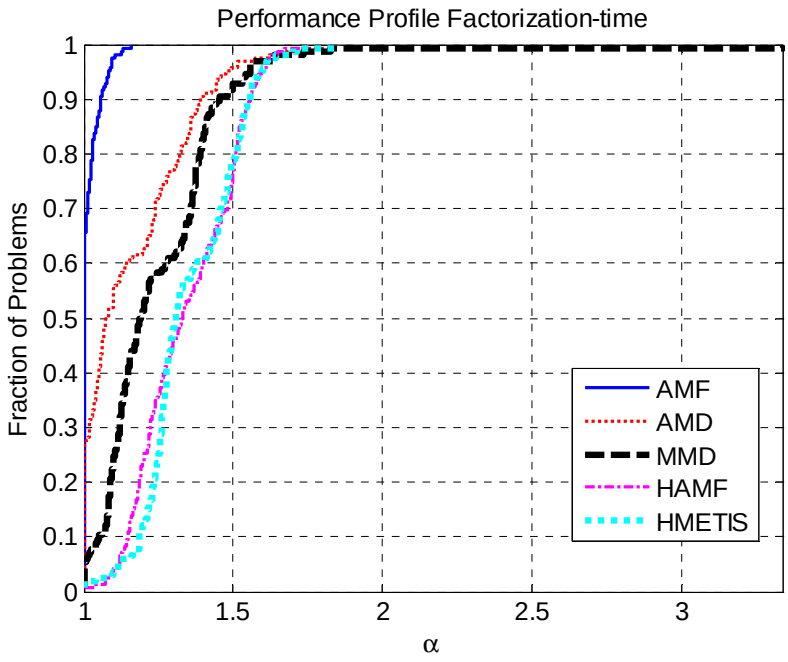


Figure 6.13: Performance profile, $p(\alpha)$: PARDISO factorization time for alternative matrix ordering programs, 166 2D test problems with regular geometries

We also perform numerical experiments on test problems with irregular geometries. Figure 6.14 shows the performance profile for the PARDISO factorization times. As shown in Figure 6.14, AMF and MMD orderings usually give the best factorization times for 2D models with irregular geometries.

Although AMF gives favorable factorization times for both regular and irregular geometries, it is known that it is the most time-consuming local ordering. We evaluate the overall execution time for factorization with alternative matrix ordering programs. Figure 6.15 and Figure 6.16 show the performance profiles for overall execution times for the 2D test problems with regular and irregular geometries respectively. As shown in Figure 6.15, AMF produces favorable overall execution times for 2D test problems with regular geometries. As shown in Figure 6.16, the performance profiles for overall execution times for AMF and MMD are similar for 2D models with regular geometries. Both AMF and MMD provide favorable overall execution times compared to other matrix ordering programs.

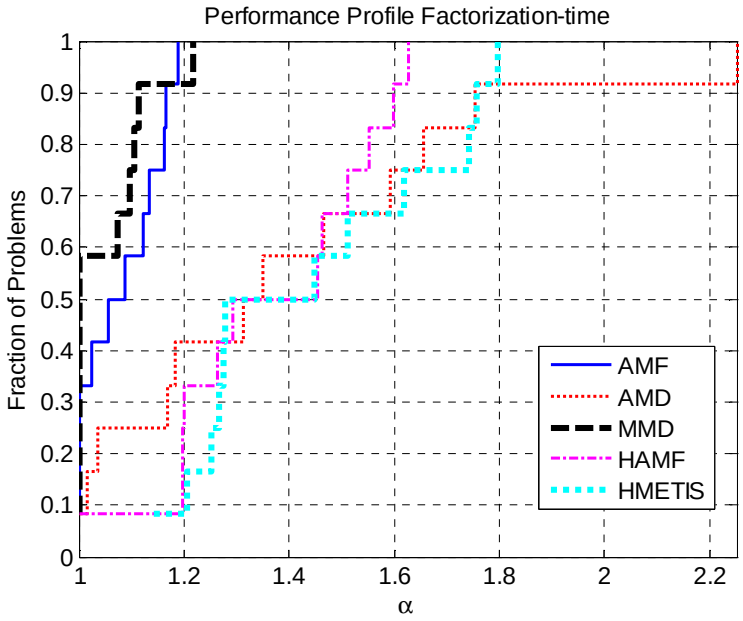


Figure 6.14: Performance profile, $p(\alpha)$: PARDISO factorization time for alternative matrix ordering programs, 42 2D test problems with irregular geometries

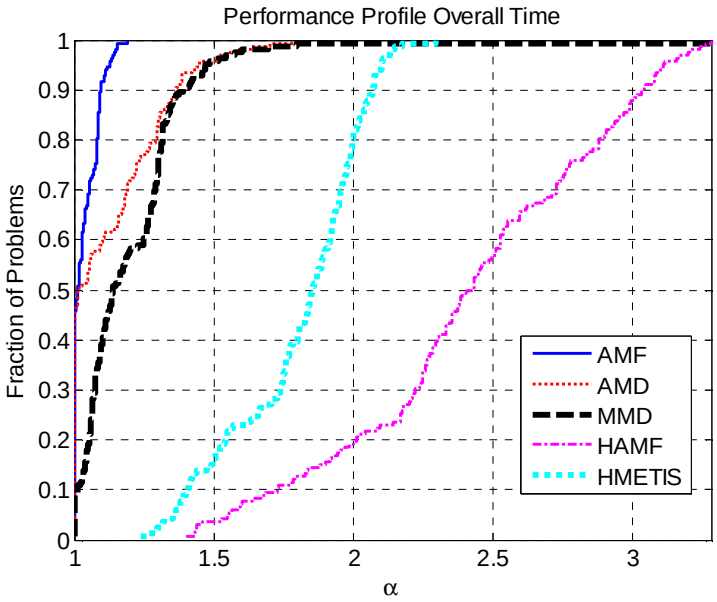


Figure 6.15: Performance profile, $p(\alpha)$: PARDISO factorization time plus matrix ordering time for alternative matrix ordering programs, 166 2D test problems with regular geometries

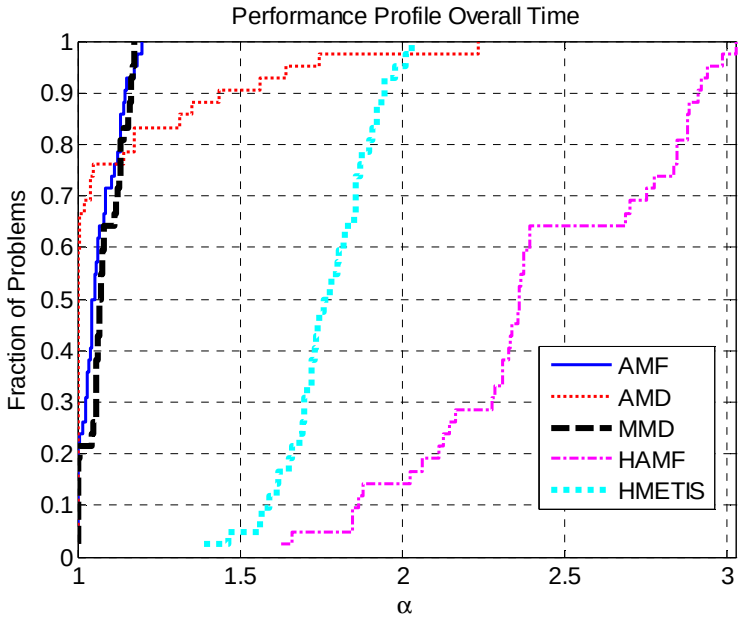


Figure 6.16: Performance profile, $p(\alpha)$: PARDISO factorization time plus matrix ordering time for alternative matrix ordering programs, 42 2D test problems with irregular geometries

Finally, we show the matrix ordering execution times for 2D test problems with regular geometries. Figure 6.17 shows the ordering times given in terms of the factorization times. The factorization times are for the pivot-ordering found by the corresponding matrix ordering program. As shown in Figure 6.17, the local orderings take significantly less time compared to the hybrid matrix ordering programs. AMF takes the longest execution time among local ordering programs. However, AMF execution time is still significantly smaller than the execution time of the hybrid matrix ordering programs. AMD was expected to give matrix ordering times better than MMD counterpart since AMD is computationally cheaper [63]. Contrary to expectations, the execution time of AMD is not better than the execution time of MMD. We believe that this is due to the use of supervariable graph. The use of supervariable graph can significantly reduce MMD execution times [129], but it does not have a significant impact on the execution time of AMD ordering [63].

As shown in Figure 6.17, the ordering time for hybrid matrix ordering programs, HAMF and HMETIS, may be larger than the factorization times for 2D test problems. In addition, HAMF generally takes more time than HMETIS. HAMF performs partitioning for two alternative random node permutations in order to produce higher quality pivot orderings. This yields pivot-orderings with fewer non-zeros but at a cost of increased execution time.

In Figure 6.17, Models 1-83 are for models with 2D quadrilateral elements and Models 84-166 are models with 2D frame elements. For the models with each element type, the model sizes usually get larger as the model number increases. As shown in Figure 6.17, the ratio of ordering time to factorization time reduces as the model size increases. In other words, the factorization time dominates the overall execution time of the solver for large 2D test problems.

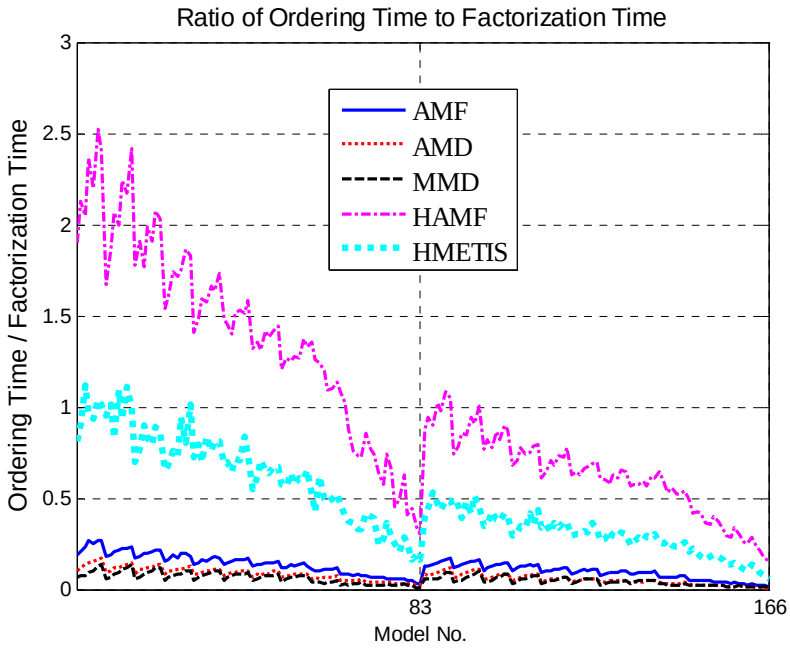


Figure 6.17: Ordering times in terms of factorization times for 2D test problems with regular geometries.

In summary, the local ordering AMF produces favorable pivot orderings with respect to non-zero and flop in reasonable times for 2D test problems. AMF can also be used to minimize the matrix ordering plus factorization times of 2D problems. For 2D test problems, the hybrid ordering programs, HAMF and HMETIS, do not offer an advantage over AMF despite their higher execution times.

6.1.3.2 3D Models

AMF proved to be efficient for 2D test problems. Among all local orderings, it yielded the most favorable flop and non-zero for the 2D test problems. However, for 3D test problems, flop and non-zero for AMF may be significantly higher than the ones for the hybrid ordering programs HAMF and HMETIS. The relative performance of the matrix ordering programs depends on the average node connectivity of the 3D problems. A similar metric, average number of elements at the columns of the coefficient matrix, is used to choose between a local ordering and hybrid ordering in the study of Duff and

Scott [58]. In our test suite, the FE models with frame elements have a lower average node connectivity value compared to the models with solid elements.

For 3D solid models with regular geometries, Figure 6.18 shows the performance profile for non-zero for three matrix ordering programs that produces favorable pivot-orderings for 3D test problems. As shown in Figure 6.18, HAMF gives the smallest non-zero for about 85% of the 3D solid models with uniform geometries. However, the performance profile for HMETIS is similar to HAMF. As shown in Figure 6.18, AMF yields significantly worse non-zero values for a majority of the 3D solid models with regular geometries.

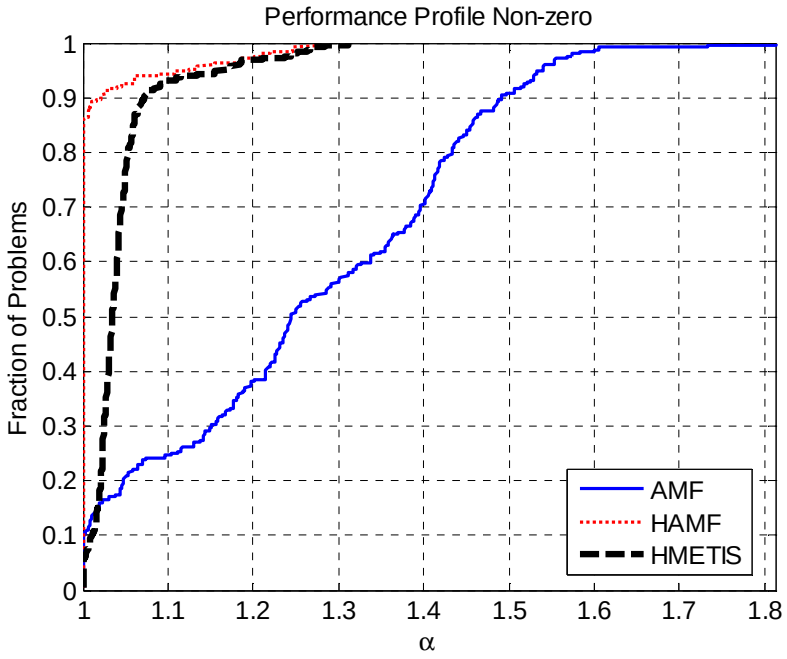


Figure 6.18: Performance profile, $p(\alpha)$: Non-zero for alternative matrix ordering programs, 252 3D solid models with regular geometries.

Similarly, Figure 6.19 shows the performance profiles for the factorization time for the 3D solid models with regular geometries. Although HAMF gives the best factorization times for the majority of the test problems, the performance of HMETIS is comparable to HAMF. As shown in Figure 6.19, AMF yields factorization times more

than 2 times the best factorization time for about 40% of the 3D solid test problems with regular geometries.

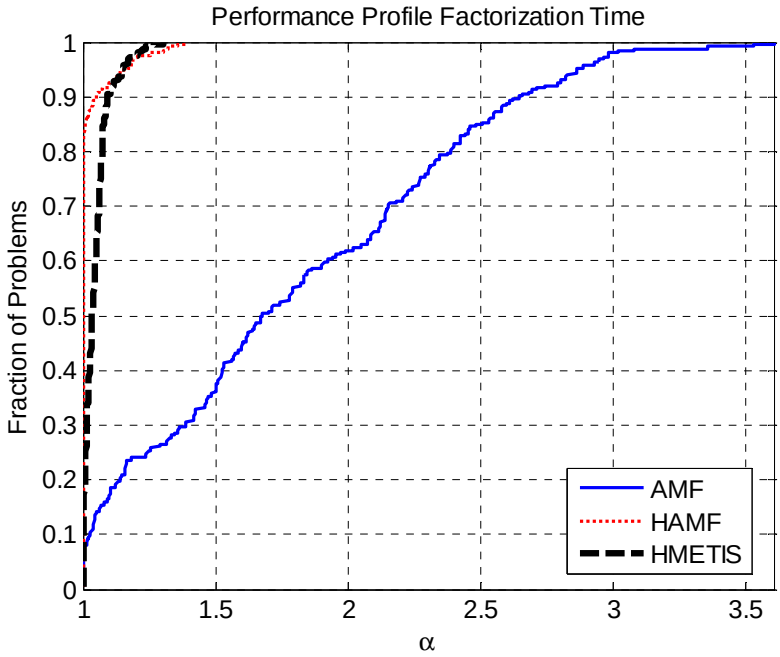


Figure 6.19: Performance profile, $p(\alpha)$: PARDISO factorization time for alternative matrix ordering programs, 252 3D solid models with regular geometries.

Next, we show the performance of the three matrix ordering programs for the 3D solid test problems with irregular geometries. Figure 6.20 and Figure 6.21 show the performance profiles for non-zero and PARDISO factorization time respectively. Similar to the test problems with regular geometries, HAMF gives the best non-zero and factorization times for the majority of the 3D solid models with irregular geometries. The performance of HMETIS is slightly worse than the HAMF as shown in Figure 6.20 and Figure 6.21.

Considering numerical experiments performed on all 3D solid models, HAMF usually yielded the most favorable non-zero and factorization time. However, as stated previously, HAMF typically required a higher execution time than HMETIS. Therefore, HAMF may not be the best option to reduce the overall time required for matrix ordering

and numerical factorization. Figure 6.22 shows the performance profile for the overall execution time for 3D solid models with irregular geometries. As shown in Figure 6.22, the overall time for HMETIS is similar to overall time for HAMF although HAMF yielded better factorization times for the test problems used for Figure 6.22.

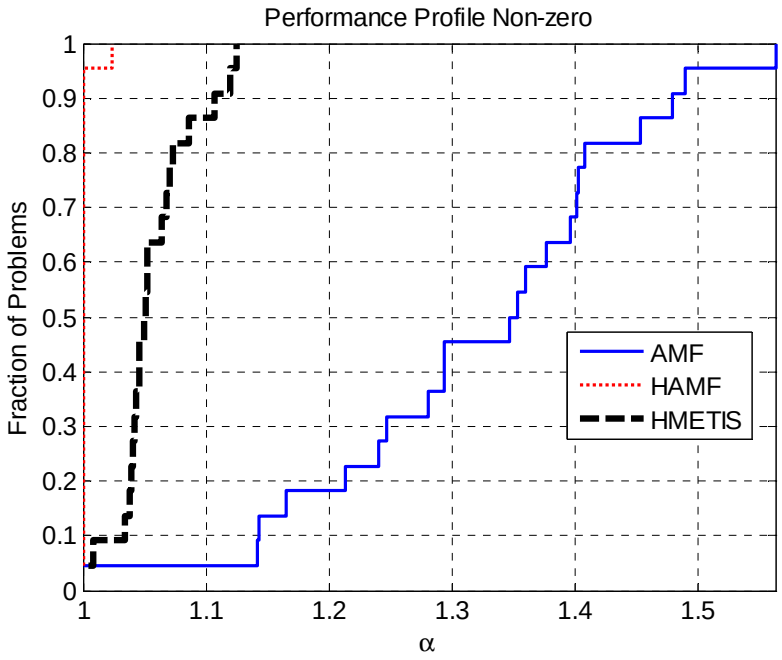


Figure 6.20: Performance profile, $p(\alpha)$: Non-zero for alternative matrix ordering programs, 22 3D solid models with irregular geometries.

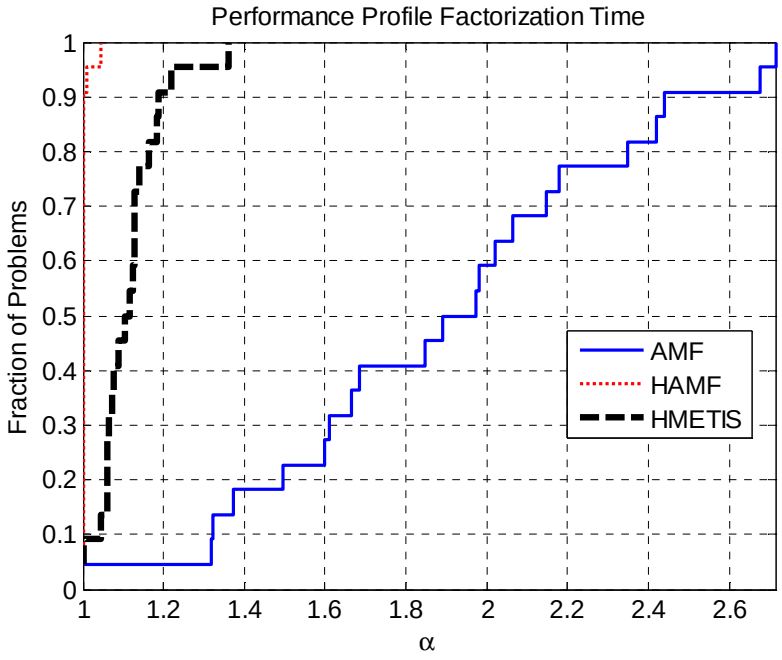


Figure 6.21: Performance profile, $p(\alpha)$: PARDISO factorization time for alternative matrix ordering programs, 22 3D solid models with irregular geometries.

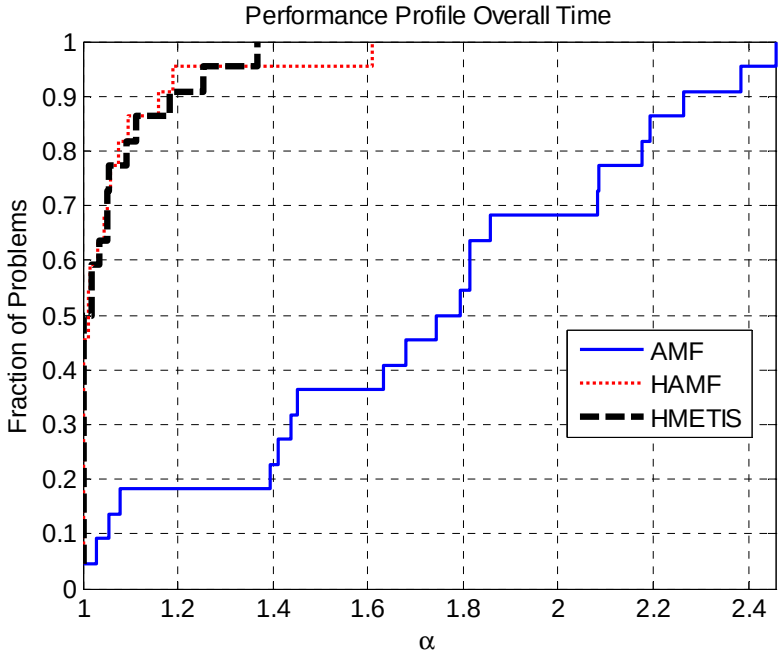


Figure 6.22: Performance profile, $p(\alpha)$: PARDISO factorization time plus matrix ordering time for alternative matrix ordering programs, 22 3D solid models with irregular geometries.

For 3D frame models, the relative performance of the three matrix ordering programs, AMF, HAMF, and HMETIS, is different than the relative performance for the 3D solid models. For 3D frame models with regular geometries, Figure 6.23 and Figure 6.24 show the performance profiles for the non-zero and PARDISO factorization times respectively. As shown in these figures, for majority of the models with 3D frame elements, AMF gives the most favorable pivot-orderings with respect to non-zero and factorization time. For the remaining 3D frame models, HMETIS gives the most favorable pivot orderings. As shown in Figure 6.24, AMF and HMETIS may yield 2 times the best factorization time.

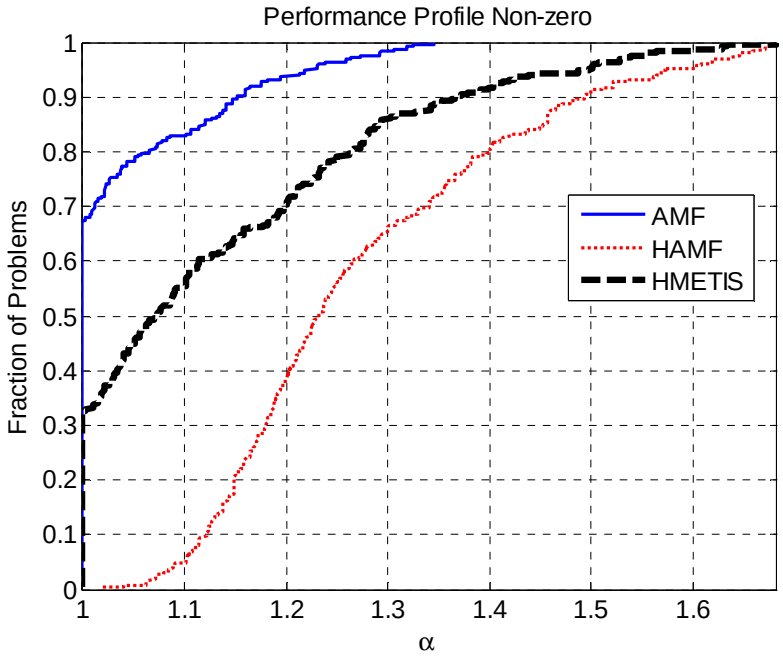


Figure 6.23: Performance profile, $p(\alpha)$: Non-zero for alternative matrix ordering programs, 22 3D frame models with irregular geometries.

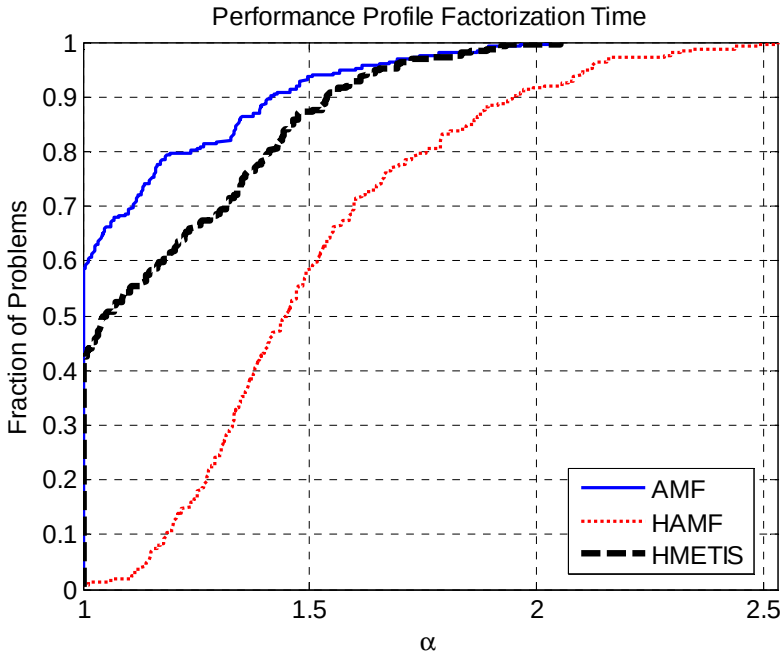


Figure 6.24: Performance profile, $p(\alpha)$: PARDISO factorization time for alternative matrix ordering programs, 22 3D frame models with irregular geometries.

The relative performance of AMF and HMETIS depends on the average node adjacency of the models. Table 6.1 shows the performance of AMF and HMETIS for regular models having average node adjacency below a certain value. The first column of Table 6.1 gives the limit for average node connectivity. Each row of Table 6.1 gives the performance of AMF and HMETIS for models having average node adjacencies less than the value given in the first column of Table 6.1. As shown in Table 6.1, as the limit for average node connectivity increases, the percentage of problems for which AMF gives the best results decreases. For example, there are 10 regular models for which the average node connectivity is below 5.4 and AMF ordering yields best factorization for all of these models. For these models, HMETIS may yield a factorization time which is 1.62× the factorization time for AMF. If the limit for the average node adjacency is taken as 5.5, AMF gives the best results for 79.49% of such models. This percentage further decreases as the average node adjacency of the models increases.

Maximum Average Node Adjacency	Number of Models	% of Problems AMF Gives the Best Factorization Time	Worst Case Factorization Time for AMF	% of Problems HMETIS Gives the Best Factorization Time	Worst Case Factorization Time for HMETIS
5.4	10	100%	1×best	0%	1.62×best
5.45	21	80.95%	1.17×best	19.05%	1.62×best
5.5	39	79.49%	1.41×best	20.51%	1.62×best
5.55	59	81.36%	1.41×best	18.64%	1.62×best
5.6	88	78.41%	1.48×best	20.45%	1.62×best
5.65	118	77.97%	1.48×best	21.19%	2.05×best
5.7	145	74.48%	1.48×best	24.83%	2.05×best
5.75	182	70.88%	1.61×best	28.57%	2.05×best
5.8	225	63.56%	2.04×best	36%	2.05×best
5.85	252	57.14%	2.04×best	42.46%	2.05×best

Table 6.1: Performance of matrix ordering programs AMF and HMETIS for 3D frame models with different average node adjacencies

For irregular geometries, the relative performance of three matrix ordering programs, AMF, METIS, and HAMF, is similar to the relative performance for the models with regular geometries. For three matrix ordering programs, Figure 6.25 and Figure 6.26 show the performance profiles for non-zeros and PARDISO factorization times respectively. Similar to regular geometries, AMF ordering gives favorable results for irregular geometries with low average node adjacencies.

Finally, Figure 6.27 shows the execution times for the three matrix ordering programs. Here, the ordering times are given in terms of the factorization times. As shown in Figure 6.27, the ordering times for hybrid orderings, HAMF and HMETIS, may be comparable to the corresponding factorization times for smaller 3D models. As we increase the model size (the model size typically increases between the model numbers shown on the x-axis of Figure 6.27), the ratio of ordering times to factorization times decreases. Since for large 3D problems, the ordering times are significantly smaller than the factorization times, trying alternative matrix ordering programs has the potential to

minimize the factorization times with a relatively small cost of additional matrix ordering times.

In summary, the hybrid ordering programs, HAMF and HMETIS, yield favorable factorization times for 3D models with solid elements. Although the HAMF yields the best factorization times for majority of the 3D solid models, the factorization times for HMETIS are similar to the ones for HAMF. AMF cannot compete with the hybrid ordering programs for 3D models with solid elements. However, AMF may yield best non-zero and factorization times for 3D models with frame elements, especially for the models with small average node adjacencies. For the remaining 3D frame models, HMETIS usually yields the best factorization times. The matrix ordering times for large 3D models are significantly smaller than the factorization times of these models. Therefore, it may be desirable to execute several matrix ordering programs in order to minimize the factorization times.

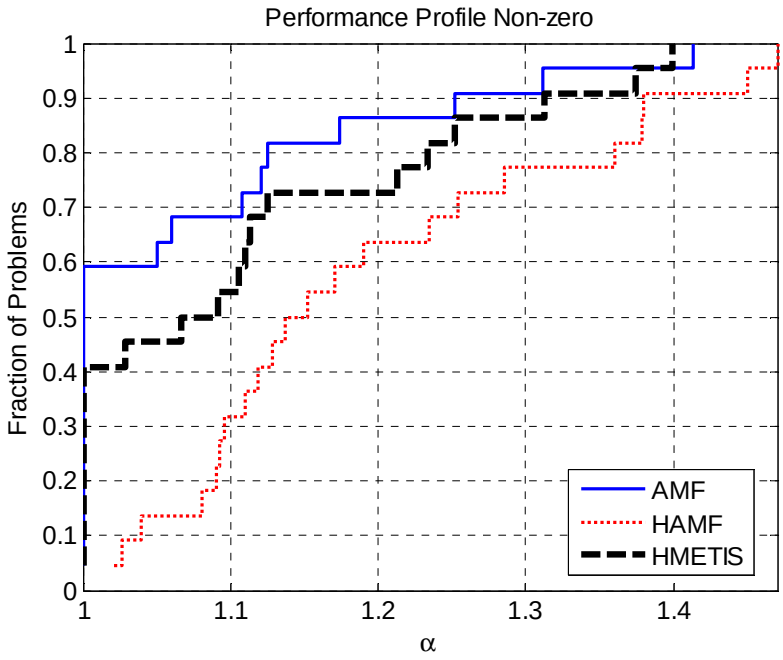


Figure 6.25: Performance profile, $p(\alpha)$: Non-zero for alternative matrix ordering programs, 22 3D frame models with irregular geometries.

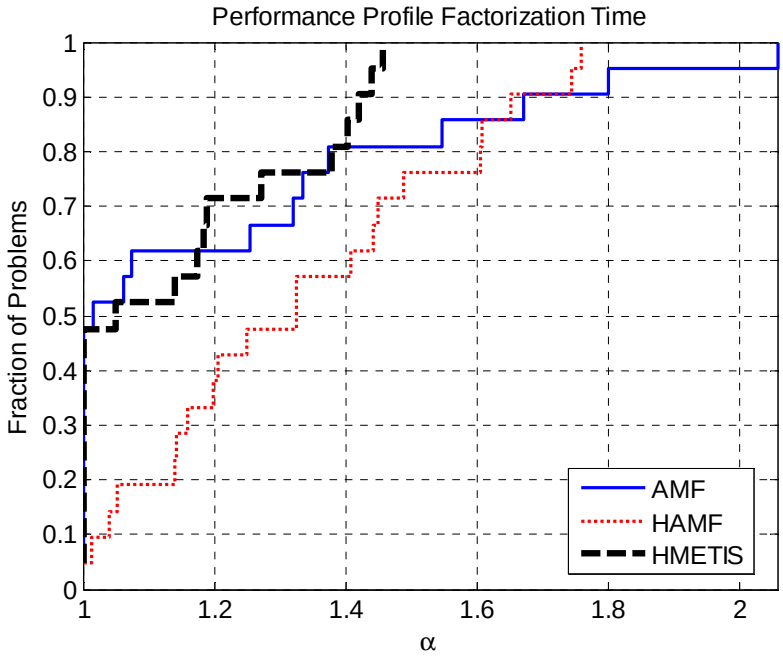


Figure 6.26: Performance profile, $p(\alpha)$: PARDISO factorization time for alternative matrix ordering programs, 22 3D frame models with irregular geometries.

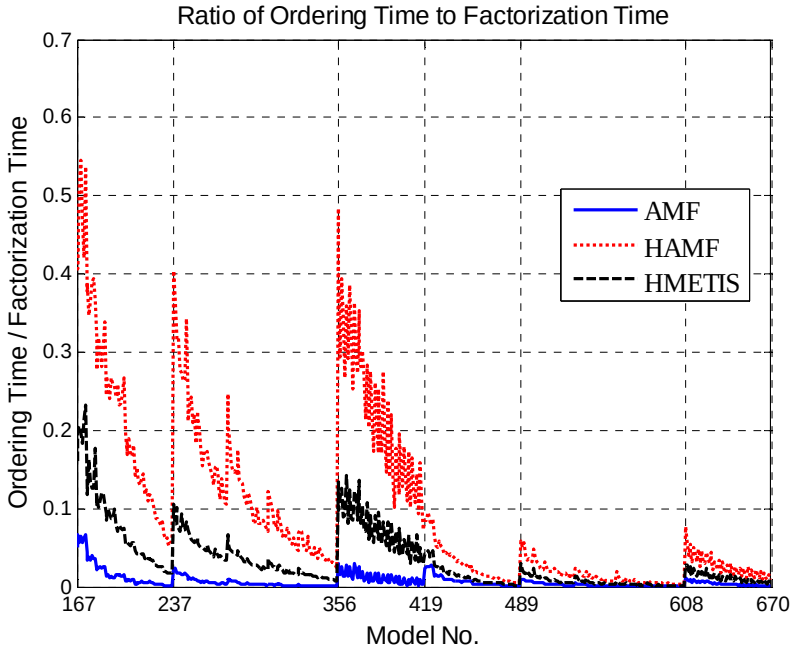


Figure 6.27: Matrix ordering time given in terms of the factorization time.

6.1.3.3 Transition between 2D and 3D

The numerical experiments show that AMF ordering performs well for 2D models. However, it may not yield favorable results for 3D models, especially for the ones with large average node adjacency. We perform numerical experiments on several test problems to determine the performance of AMF for 3D models with 2D-like geometries, in other words, 3D models with a small number of elements in the third dimension. We do a preliminary study to determine the threshold for number of elements in the third dimension after which the hybrid ordering HMETIS becomes the better choice. Figure 6.28 and Figure 6.29 show some performance parameters for AMF ordering normalized according to the performance parameters for HMETIS. Figure 6.28 is for models with quadrilateral and solid elements. The number of elements in x and z directions are the same for all models and only the number of elements in y direction changes. As shown in Figure 6.28, the relative AMF performance is similar for models with up to 6 elements in y direction (we consider s100×4×200 as an exception). For s100×6×200, AMF factorization time and flop are significantly worse than HMETIS counterparts. Therefore, we consider 6 elements in third dimension as a threshold value after which the performance of AMF becomes worse than HMETIS. Figure 6.29 shows the relative performance of AMF for 2D and 3D frame models. Similar to the results given in Figure 6.28, the relative performance of AMF decreases as the number of elements in y direction approaches 6.

The numerical experiments in this section illustrate that the AMF has the potential to reduce the factorization time and non-zero for 3D models with 2D-like geometries. More numerical experiments are required to provide more accurate threshold values in order to determine 2D-like models for which the use of AMF may be advantageous. The extension of AMF efficiency to 2D-like geometries is a desirable feature since AMF has significantly smaller execution times compared to the hybrid matrix ordering programs.

The execution time of the hybrid ordering programs is significant compared to the factorization time for 2D and 2D-like models.

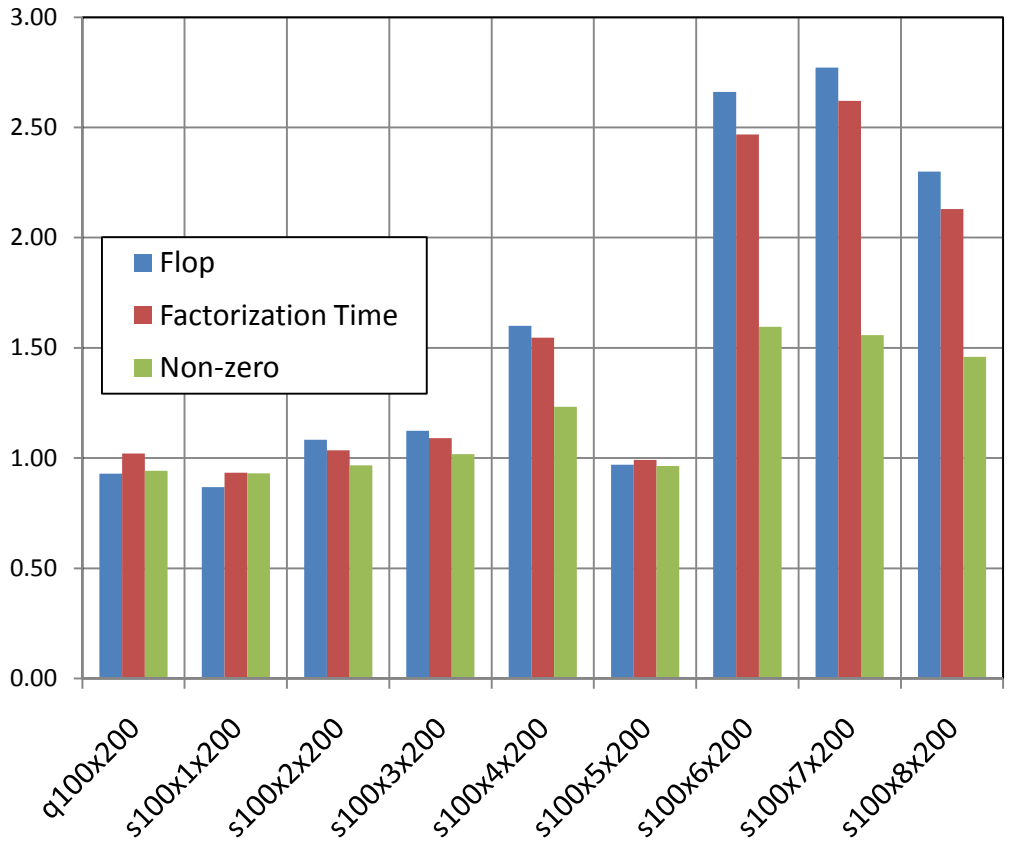


Figure 6.28: Performance parameters for AMF normalized according to the results of HMETIS for 2D and 2D-Like models with quadrilateral and solid elements respectively

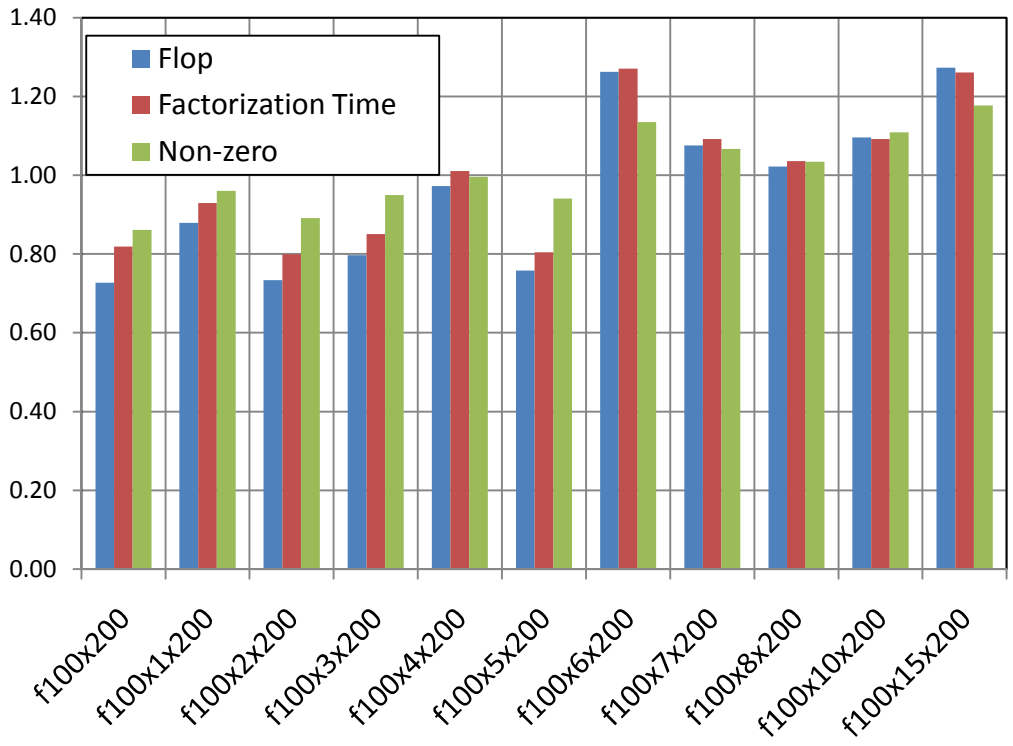


Figure 6.29: Performance parameters of AMF normalized according to the results of HMETIS for 2D and 2D-Like models with 2D frame and 3D frame elements respectively

6.1.4 Matrix Ordering for Parallel Factorization

As discussed in the previous sections, the local ordering AMF may yield favorable serial factorization times for certain models. However, the local orderings typically produce long and unbalanced assembly trees, which reduce the amount of tree-level parallelism that can be exploited for parallel factorization [74, 77, 136]. Our mapping algorithm automatically chooses between the tree level parallelism and matrix level parallelism. Even though the assembly tree corresponding to a local ordering is not suitable for tree-level parallelism, current SMP architectures may allow obtaining satisfactory parallel performance by primarily exploiting the matrix level parallelism.

Therefore, a pivot-ordering with a local ordering may still give a better multithreaded factorization time compared to the hybrid ordering.

Numerical experiments for serial factorization show that AMF minimizes the factorization times for majority of 2D models and HMETIS produces favorable factorization times for 3D models. In this section, we investigate whether the relative serial factorization time for AMF and HMETIS changes for parallel factorization. Figure 6.30 shows the ratio of AMF factorization time to HMETIS factorization time for serial and parallel factorization with SES solver package. If the ratio is less than one, then the factorization for AMF ordering is faster than the factorization with HMETIS ordering. As shown in Figure 6.30, the AMF to HMETIS factorization time ratios typically increase for the multithreaded factorization. The increase in the factorization time ratios indicates that the HMETIS matrix ordering improves the performance of four-thread factorization even though it may not be the best choice for the serial factorization. The relative performance increase is dramatic for some models such as Models 1, 2, 11, 12, 21, 31, and 39. Compared to the serial factorization, the number of models for which AMF gives the best factorization times reduces for the multithreaded factorization. For the serial factorization, AMF gives the best factorization times for 23 models. On the other hand, for the multithreaded factorization, AMF gives the best factorization times for 11 models. HMETIS is now the better alternative for the multithreaded factorization of the 12 models for which the AMF ordering yields better serial factorization times.

The next question to be answered is whether the matrix ordering program that produces better multithreaded factorization times can be predetermined prior to the numerical factorization. It is hard to determine which matrix ordering program will perform better merely based on the properties of the FE model since the multithreaded factorization times depend on the assembly tree structure, which is found after the analysis phase. Nevertheless, the parallel factorization time predictions computed in the analysis phase can be used to estimate the best pivot-ordering among the results from

alternative matrix-ordering programs. Figure 6.31 shows the estimated and actual factorization performance of AMF relative to HMETIS for four-thread factorization. As shown in Figure 6.31, the relative performance estimates are inaccurate for some models. This is mainly due to the inaccuracies in the predicted parallel factorization times and further discussed in the subsequent Chapter. Nevertheless, the factorization time estimations usually predict which matrix ordering will give the best parallel factorization times for the problems in the benchmark suite. Figure 6.32 shows the performance profiles for choosing the pivot-ordering among the results of HMETIS and AMF based on the serial and parallel estimated factorization times. Figure 6.32 also shows the performance profile for the best factorization time, which intersects with the y-axis. As shown in Figure 6.32, the strategy that chooses the pivot-orderings based on the estimated parallel factorization times usually gives the best factorization times. Therefore, multithreaded factorization time predictions in the analysis phase can be used to choose among alternative pivot-orderings with different matrix ordering programs. The performance profiles also show that choosing the pivot-ordering based on the estimated serial factorization times may lead to multithreaded factorization times up to 1.9 times the best factorization time.

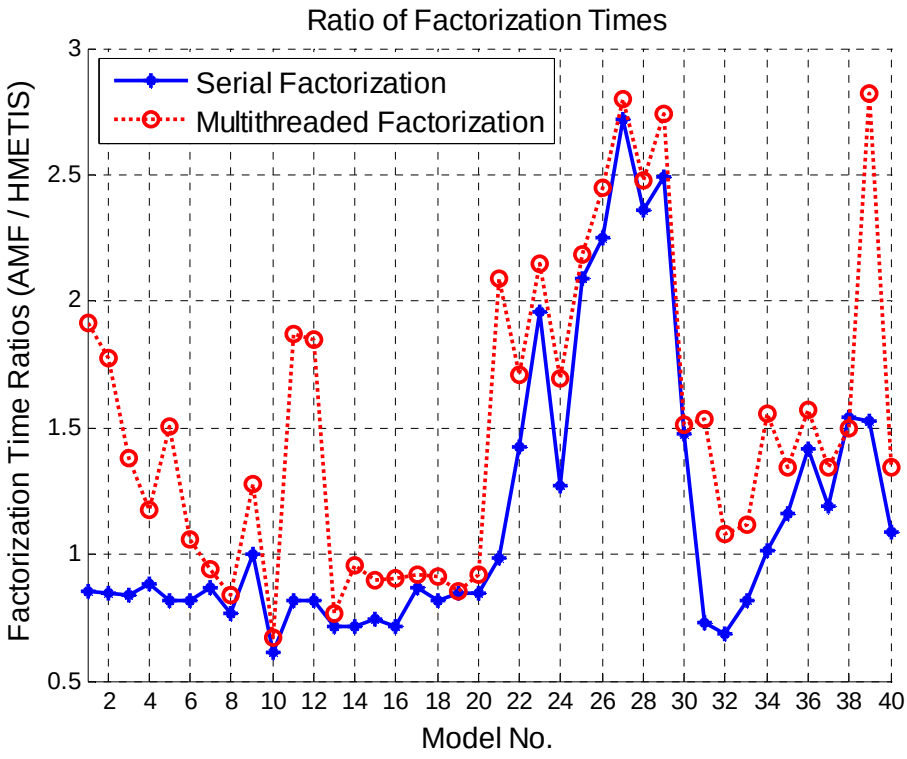


Figure 6.30: AMF factorization times relative to HMETIS factorization times for serial and multithreaded numerical factorization, benchmark suite of 40 test problems.

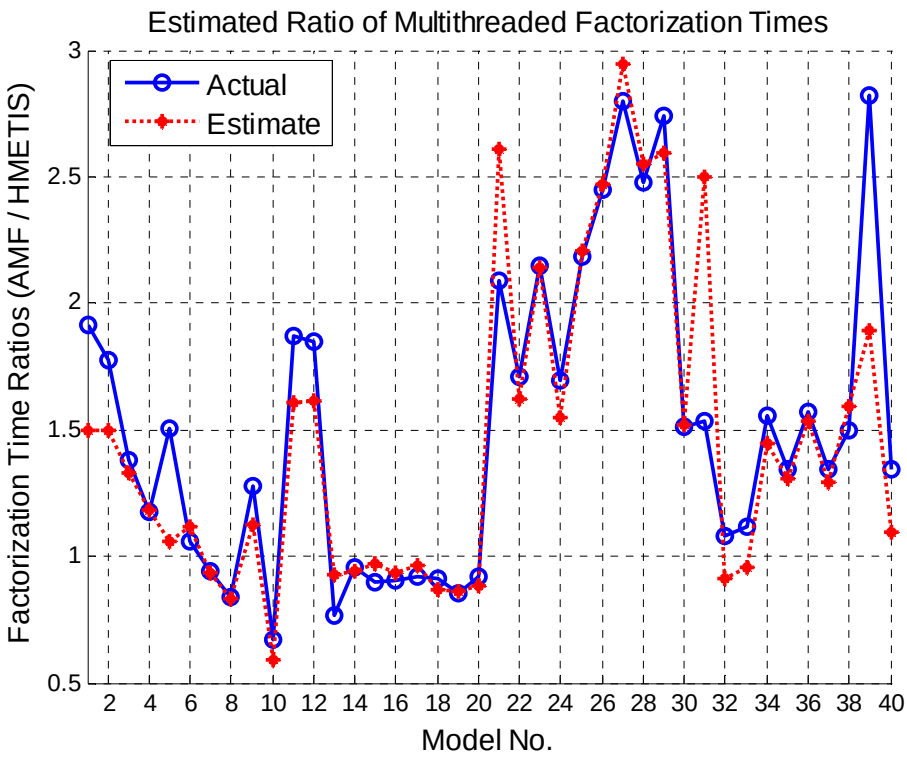


Figure 6.31: Estimated four-thread factorization times for AMF relative to the four-thread factorization times for METIS, benchmark suite of 40 test problems.

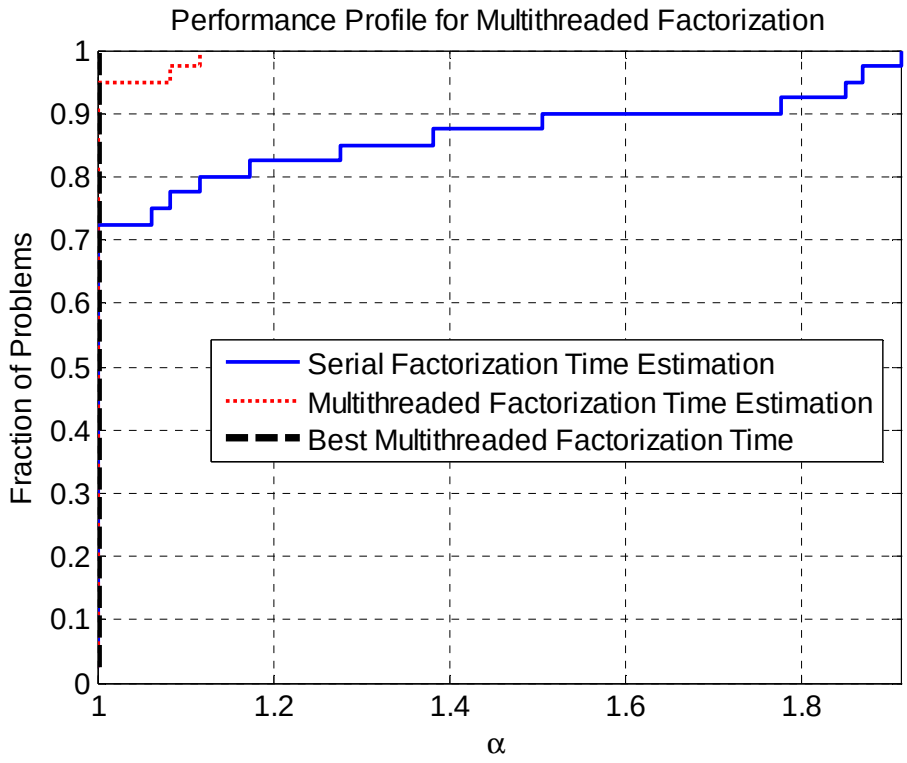


Figure 6.32: Multithreaded factorization performance profiles of alternative strategies for choosing the best pivot-ordering among the results of AMF and HMETIS, benchmark suite of 40 test problems.

6.2 Execution Time of Analysis Phase

The execution time of the PARDISO analysis phase is compared with the execution time for the PARDISO factorization. The analysis phase in PARDISO also includes time required for memory allocation for the factorization phase. However, the memory allocation time is insignificant compared to the total analysis time. Figure 6.33 shows the analysis times normalized according to the factorization times for 670 test problems with regular geometries (see Section 2.6 for a detailed description of the test problems with regular geometries). The analysis time can be larger than the factorization time for test problems which have small factorization time (smaller than 0.25 sec). The analysis time is usually a fraction of the factorization time for the test problems having factorization times larger than 5.0 sec.

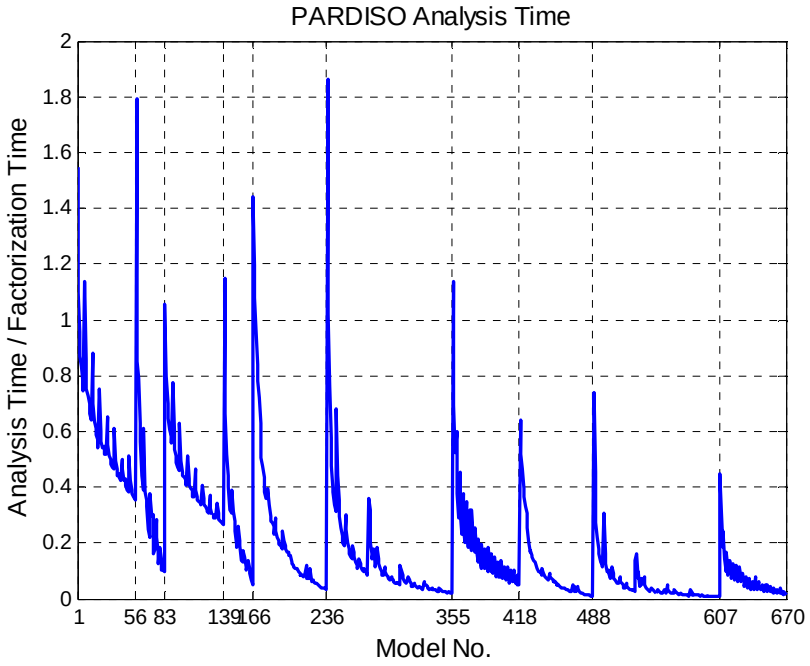


Figure 6.33: Analysis time divided by the factorization time for PARDISO.

The number of dofs in a FE model is found to affect the execution time of the analysis phase relative to the execution time of the factorization phase. Figure 6.34 shows the relationship between the normalized analysis time and number of dofs for 2D test problems. As shown Figure 6.34, as the number of dofs increases the relative execution time of the analysis phase decreases. For 2D problems having less than 30,000 dofs, the analysis phase takes more than 20% of the time required for the factorization. Figure 6.35 shows the same plot as Figure 6.34 for 3D test problems. As shown in Figure 6.35, the analysis time is less than 20% of the factorization time for the 3D test problems having more than 6,000 dofs. For 3D problems having more than 30,000 dofs, the analysis time is less than 10% of the factorization time. Comparing Figure 6.34 and Figure 6.35, the analysis phase typically takes more time (relative to the factorization time) for a 2D test problem compared to a 3D test problem if the 2D FE model and the 3D FE model have the same number of dofs.

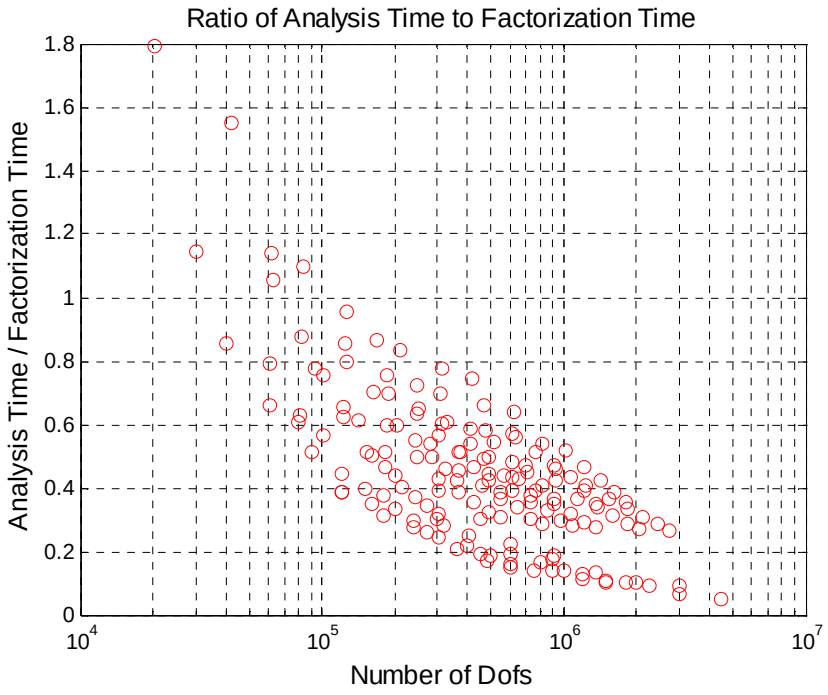


Figure 6.34: Relationship between the number of dofs and relative PARDISO analysis time for 2D test problems with regular geometries.

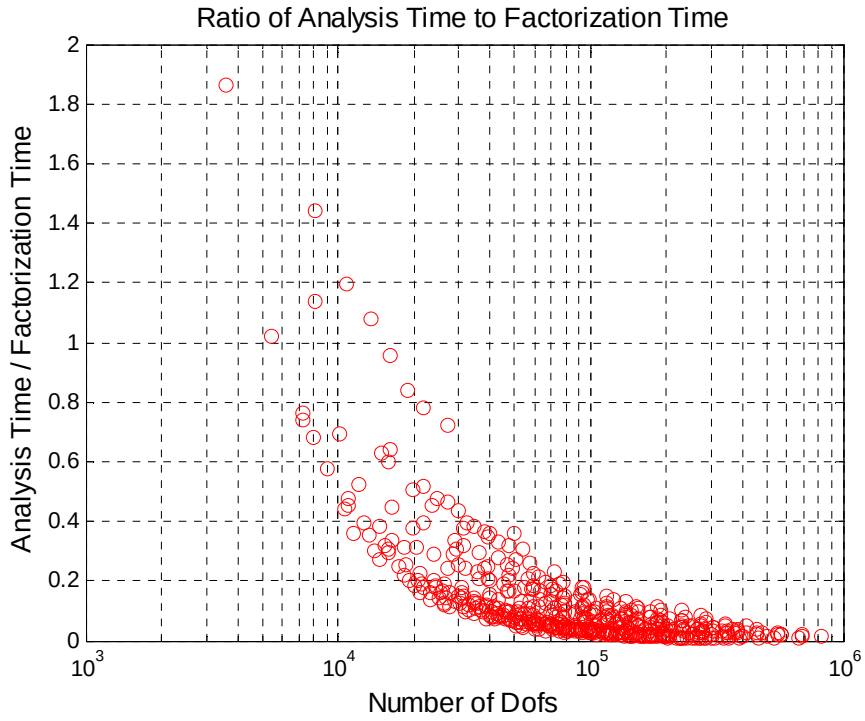


Figure 6.35: Relationship between the number of dofs and relative PARDISO analysis time for 3D test problems with regular geometries.

6.3 Optimal Coarsening

The optimal amount of coarsening depends on the dimensionality of the model. For 2D models, multiple nodes can be eliminated at each super-element formed by merging adjacent elements in the original model. On the other hand, for 3D models, the non-zero and flop increase dramatically if we perform an aggressive coarsening which eliminates several nodes at each super-element. The parameters *nodeco* and *eleco* control the amount of coarsening for element based and node based coarsening respectively. Numerical experiments are performed with different *nodeco* and *eleco* values to evaluate the efficiency of the coarsening schemes for the HMETIS and AMF matrix ordering programs.

For 2D problems in the benchmark suite of 40 test problems, Figure 6.36 shows the performance profiles for factorization with different *nodeco* values. This figure shows the performance profiles for *nodeco* = 1, 4, 8, and 9. These are the *nodeco* values that may yield the best factorization times. The pivot-ordering is found by HMETIS for the results shown in Figure 6.36. As shown in Figure 6.36, *nodeco*=1 gives best performance profile for the factorization with HMETIS orderings. The performance profiles for *nodeco* = 1, 8, and 9 shown in Figure 6.36 are similar. The use of original mesh may yield factorization times 1.6 times the best factorization time as shown in Figure 6.36. This worst case factorization time is for the test problem q500×1500. For q500×1500, the use of original mesh yield flop values almost 2 times the flop values for the node based coarsening with *nodeco*=1. Consequently, the factorization time for the original mesh is significantly larger than the factorization time for the coarsened mesh.

The impact of coarsening on HMETIS matrix ordering times is shown Figure 6.37 for 2D test problems. As shown in Figure 6.37, for coarsening schemes with *nodeco* = 8 and 9, HMETIS takes significantly smaller time compared to the execution time for the original mesh. The coarsening scheme with *nodeco*=8 makes HMETIS run about 3 times

faster for half of the 2D problems in the benchmark suite. The less aggressive coarsening schemes such as *nodeco*=1 and *nodeco*=2 also reduce the HMETIS ordering times.

The element based coarsening also improves the factorization times for 2D problems processed with HMETIS. For the element based coarsening, improvements in the factorization times are similar to the improvements for the node based coarsening. For 2D problems, the use of *eleco*=1 and *eleco*=4 usually yields favorable factorization times for the HMETIS ordering.

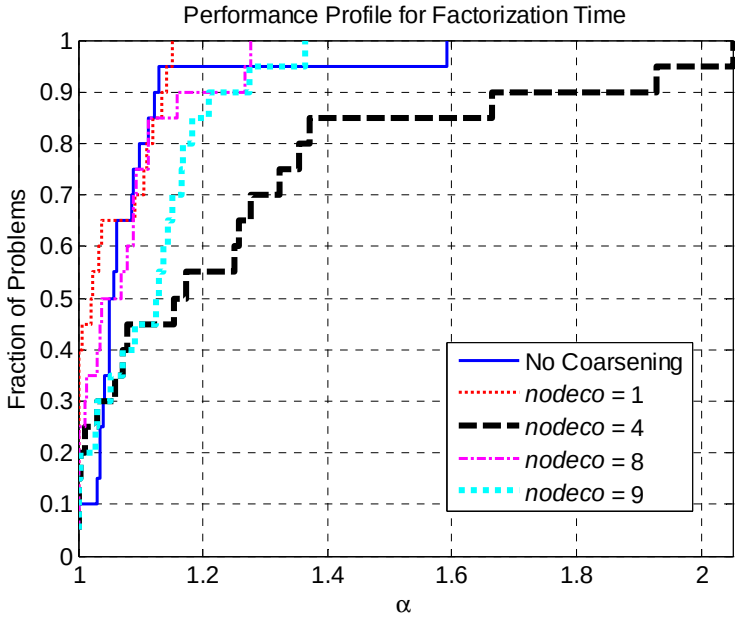


Figure 6.36: For 2D problems, performance profiles for factorization times with alternative *nodeco* values, HMETIS ordering. 2D models in the benchmark suite of 40 test problems are used.

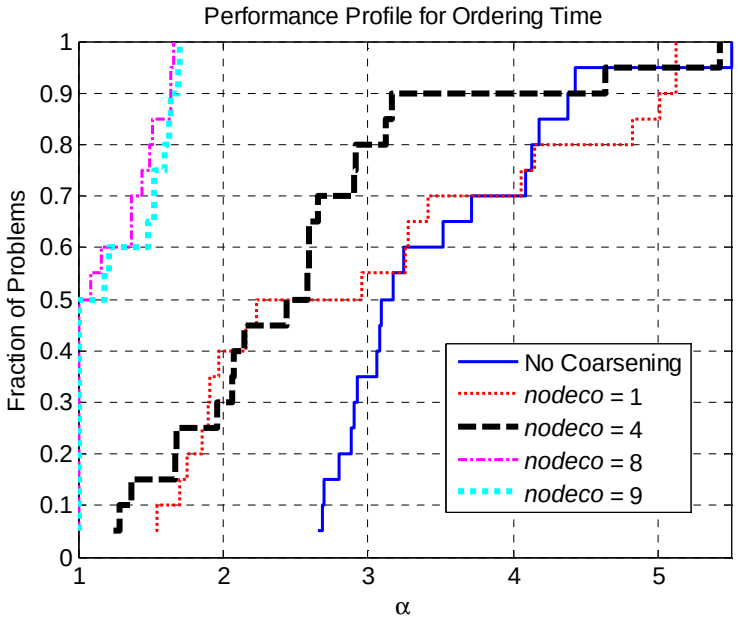


Figure 6.37: For 2D problems, performance profile for matrix ordering times with alternative *nodeco* values, HMETIS ordering. 2D models in the benchmark suite of 40 test problems are used.

The effect of coarsening for the local ordering AMF is also investigated for 2D problems since AMF ordering usually yields favorable serial factorization times for 2D problems. Figure 6.38 shows the factorization times for alternative *nodeco* values for AMF matrix ordering. As shown in Figure 6.38, *nodeco*=1 provides favorable factorization times for AMF matrix ordering. The factorization times for *nodeco*=4, 8, and 9 are significantly worse than the alternatives, *nodeco*=0 and 1, as shown in Figure 6.38. For 2D test problems, Figure 6.39 compares the AMF matrix ordering times for different *nodeco* values. As shown in Figure 6.39, coarsening scheme with *nodeco*=1 can reduce the matrix ordering times significantly. For 2D problems, the element based coarsening usually gives factorization times worse than the factorization times for the original mesh for the matrix ordering program AMF.

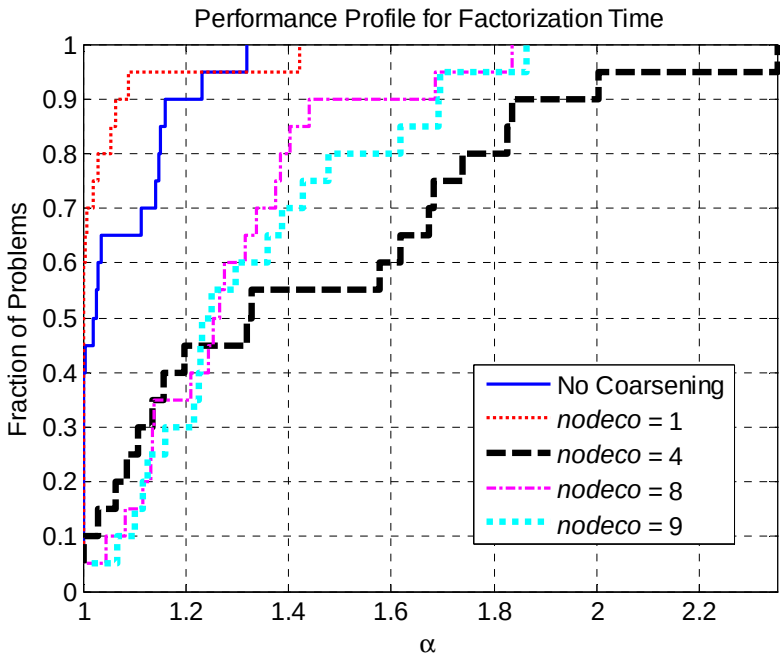


Figure 6.38: For 2D problems, performance profile for factorization times with alternative *nodeco* values, AMF ordering. 2D models in the benchmark suite of 40 test problems are used.

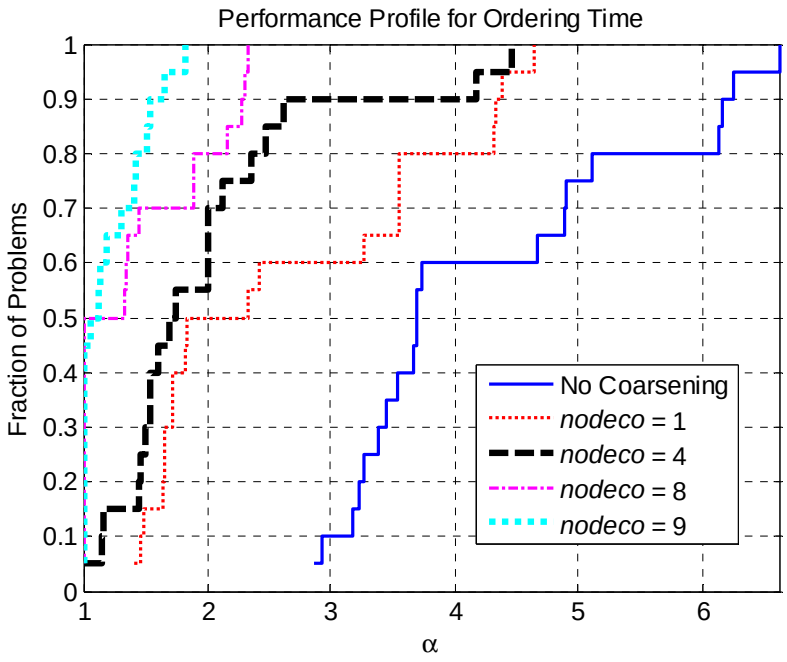


Figure 6.39: For 2D problems, performance profile for matrix ordering times with alternative *nodeco* values, AMF ordering. 2D models in the benchmark suite of 40 test problems are used.

According to the numerical experiments on 2D problems, *nodeco*=1 can minimize the factorization times for both HMETIS and AMF. *nodeco*=1 can also reduce the matrix ordering times, especially for AMF matrix orderings. For 2D problems, an aggressive coarsening scheme, i.e., *nodeco*=8, can be used with HMETIS ordering. The HMETIS matrix ordering times reduce significantly with the use of *nodeco*=8. The use of *nodeco*=8 will give factorization times similar to the ones for the original mesh for the HMETIS matrix ordering. However, the use of *nodeco*=8 may increase the factorization times significantly for the AMF matrix ordering.

Next, we investigate the factorization and matrix ordering times for coarsening schemes with alternative *nodeco* and *eleco* values for 3D problems. Figure 6.40 shows the performance profile for factorization times for pivot-ordering found with the HMETIS matrix ordering program. As shown in Figure 6.40, *nodeco*=1 and original

mesh yields favorable factorization times. More aggressive coarsening schemes usually increases the factorization times. As shown in Figure 6.40, even the element based coarsening with *eleco*=1 increases the factorization times for ordering 3D problems with HMETIS matrix ordering. Figure 6.41 shows the matrix ordering times for alternative *nodeco* and *eleco* values. As shown in Figure 6.41, the node based coarsening scheme with *nodeco*=1 do not reduce the matrix ordering times significantly for 3D problems ordered with the HMETIS matrix ordering. Element based coarsening scheme may reduce the factorization times as shown in Figure 6.41. However, as shown in Figure 6.40, the factorization performance is typically worse than the one for the original mesh for the element based coarsening.

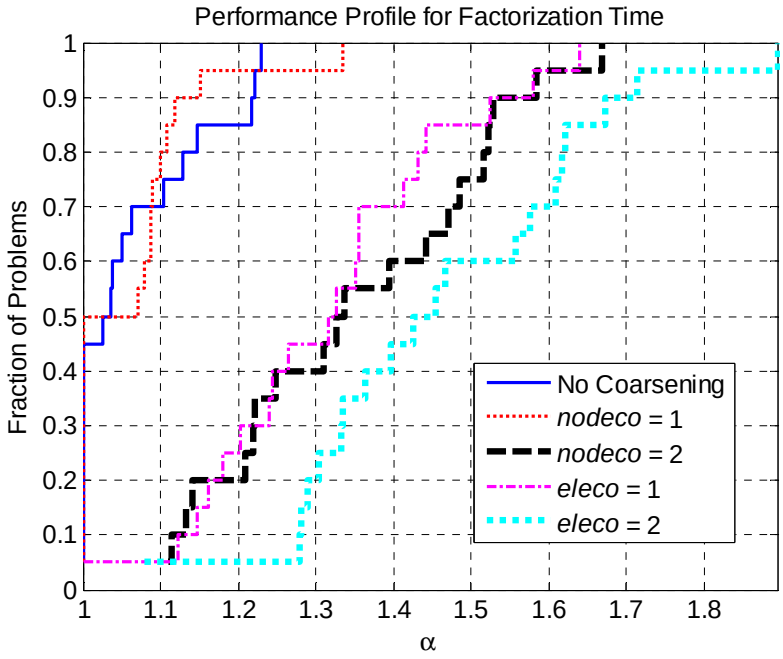


Figure 6.40: For 3D problems, performance profile for factorization times with alternative *nodeco* and *eleco* values, HMETIS ordering. 3D models in the benchmark suite of 40 test problems are used.

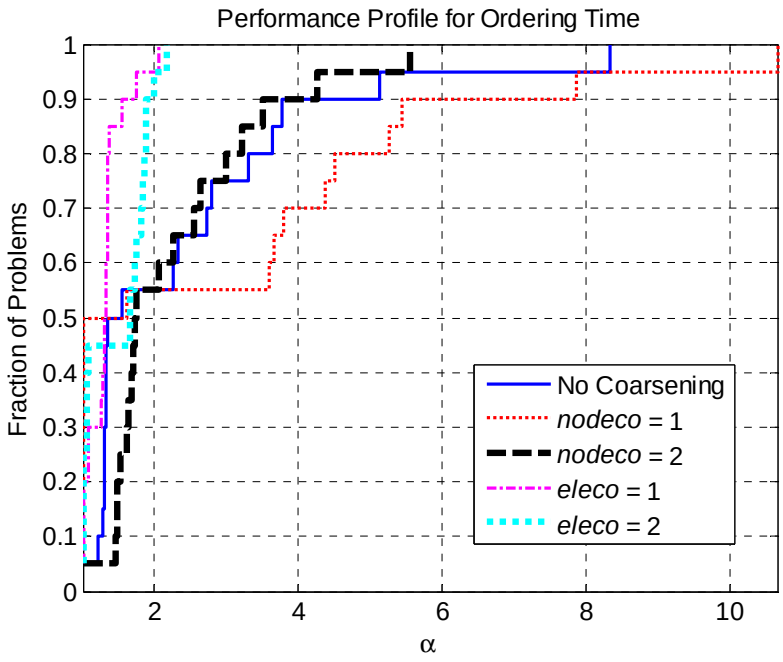


Figure 6.41: For 3D problems, performance profile for matrix ordering times with alternative *nodeco* and *eleco* values, HMETIS ordering. 3D models in the benchmark suite of 40 test problems are used.

The numerical experiments show that the coarsening scheme significantly reduces the matrix ordering times for the HMETIS hybrid matrix ordering program used for 2D FE models. This is a desirable feature since the execution time of the hybrid matrix ordering programs are comparable to the factorization times for 2D FE models as previously demonstrated in Section 6.1.3.1. The matrix ordering time reductions are significant for *nodeco*=8. Furthermore, for coarsening with *nodeco*=8, the factorization times are comparable to the factorization times for the original mesh. Therefore, the matrix ordering plus numerical factorization times can be minimized by using *nodeco*=8 for 2D FE models ordered with HMETIS hybrid ordering. For 3D test problems ordered with the HMETIS matrix ordering program, the *nodeco* and *eleco* values that provide a significant reduction in matrix ordering times usually increases the numerical factorization times. However, for 3D models, reducing the matrix ordering times is not as

crucial as it is for 2D models since the matrix ordering times are typically smaller than the numerical factorization times for 3D models.

The performance profiles for coarsening schemes show that there is no single value for *eleco* or *nodeco* parameters that gives the best factorization times. The coarsening schemes with different *nodeco* or *eleco* values minimize the factorization times for different FE models. We can use this property to construct a matrix ordering strategy to minimize the factorization time for a FE model. In this strategy, for an input FE model, matrix ordering program is executed for alternative coarsened meshes and the pivot-ordering that is expected to yield the best factorization performance is used for the numerical factorization. Next, we illustrate the performance of such a strategy that chooses the best pivot-ordering based on the flop values calculated for alternative coarsened meshes.

For the AMF matrix ordering, Figure 6.42 shows factorization time performance profile for the strategy that chooses the best pivot-ordering and performance profile for using the original mesh. As shown in Figure 6.42, for the AMF ordering, the factorization times can be improved by a factor of 2.7 by choosing the pivot-ordering that gives the minimum flop. The dramatic improvements in the factorization times shown in Figure 6.42 are mainly for the 3D test problems.

For the HMETIS matrix ordering, Figure 6.43 shows factorization time performance profile for the strategy that chooses the best pivot-ordering and performance profile for using the original mesh. As shown in Figure 6.43, choosing the best pivot-ordering strategy can improve the factorization times for about 60% of the test problems. The reductions in factorization times are modest except from a single test problem for which the factorization becomes 1.6 times faster than the use of original mesh for HMETIS. For HMETIS ordering, trying more alternatives for *nodeco* and *eleco* parameters offers little improvement in the factorization times, especially for 3D problems.

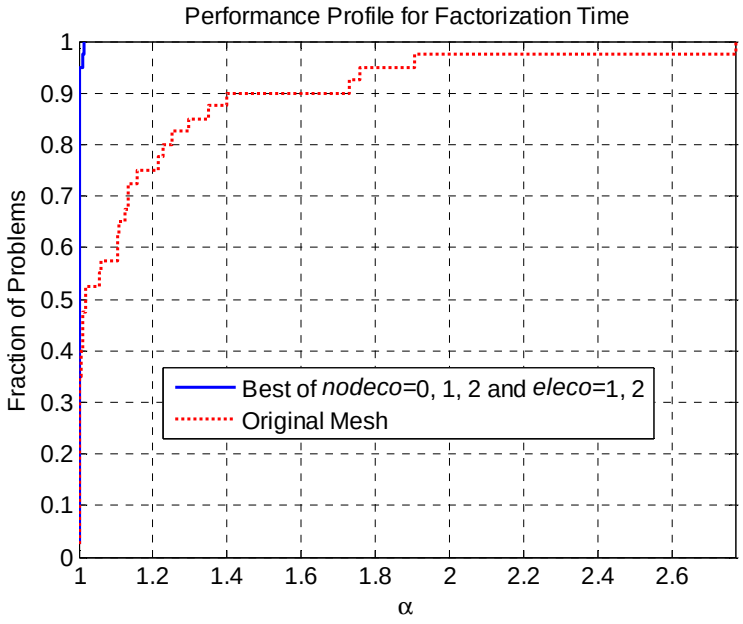


Figure 6.42: Performance profile for factorization time for choosing the best pivot-ordering among coarsened and original meshes. Pivot-orderings are found with AMF. The results are for benchmark suite of 40 test problems.

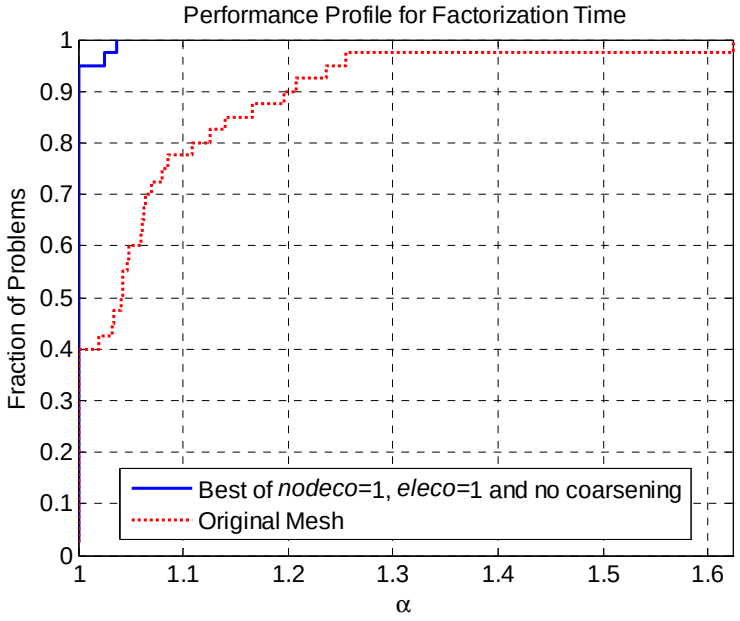


Figure 6.43: Performance profile for factorization time for choosing the best pivot-ordering among coarsened and original meshes. Pivot-orderings are found with HMETIS. The results are for benchmark suite of 40 test problems.

6.4 Optimal Node Amalgamation

The optimal value for *smin* is investigated in order to minimize the factorization time. See Section 6.4 for the description of the node amalgamation parameter, *smin*. For the results presented in this section, node blocking is applied for the node amalgamation parameter *blkmin* is set to 50.

Figure 6.44 shows the performance profile for the factorization times for various *smin* values. In Figure 6.44, the performance profile for *smin*=0 represents factorization without node amalgamation. Figure 6.44 shows that using *smin*=25 gives the most favorable factorization times. The factorization performance slightly degrades for *smin* values larger than 25. However, the performance profiles are similar for *smin* values larger than 10 as shown in Figure 6.44. As shown in Figure 6.44, the factorization times without node amalgamation may be larger than 1.6 times the factorization times with an optimal node amalgamation.

Figure 6.45 shows the ratio of factorization times with and without node amalgamation. Figure 6.46 shows the ratio of update operations to factorization operations for multifrontal factorization with and without node amalgamation. As shown in Figure 6.45 and Figure 6.46, the node amalgamation is especially useful for 2D problems for which the ratio of update operations to factorization operations are large. The ratio is large especially for smaller 2D problems. As this ratio decreases, the performance gains due to node amalgamation become less significant. Figure 6.46 clearly demonstrates the effect of node amalgamation for the problems in the benchmark suite. It increases the number of floating point operations performed for each update matrix assembly operation. In a way, it improves the numerical factorization performance by increasing computations per memory access ratio. As illustrated previously, a high value of computations per memory access is desired for the modern processors with memory hierarchies. As shown in Figure 6.45, compared to no node amalgamation, the node

amalgamation never increases factorization times for the test problems in the benchmark suite.

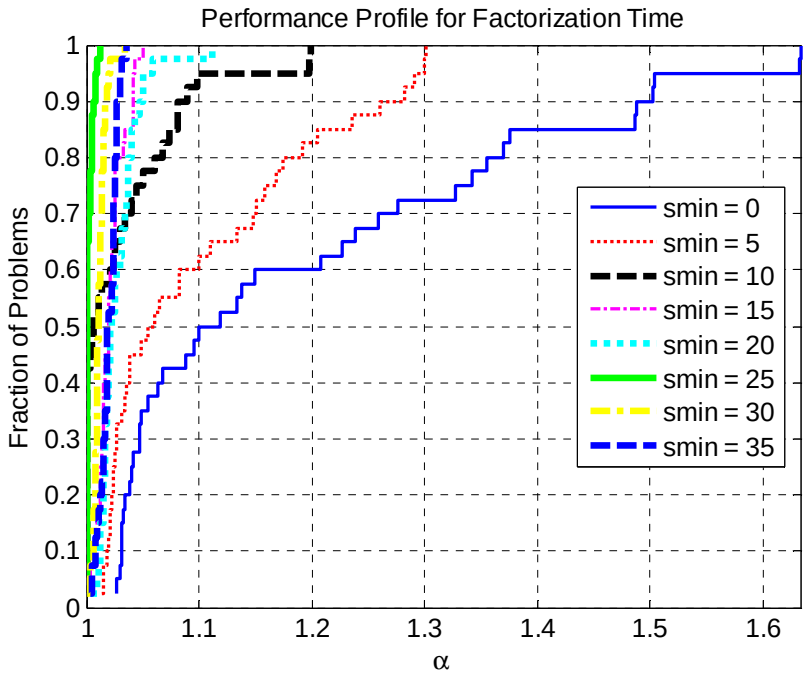


Figure 6.44: For various smin values, performance profile for the factorization time, benchmark suite of 40 test problems.

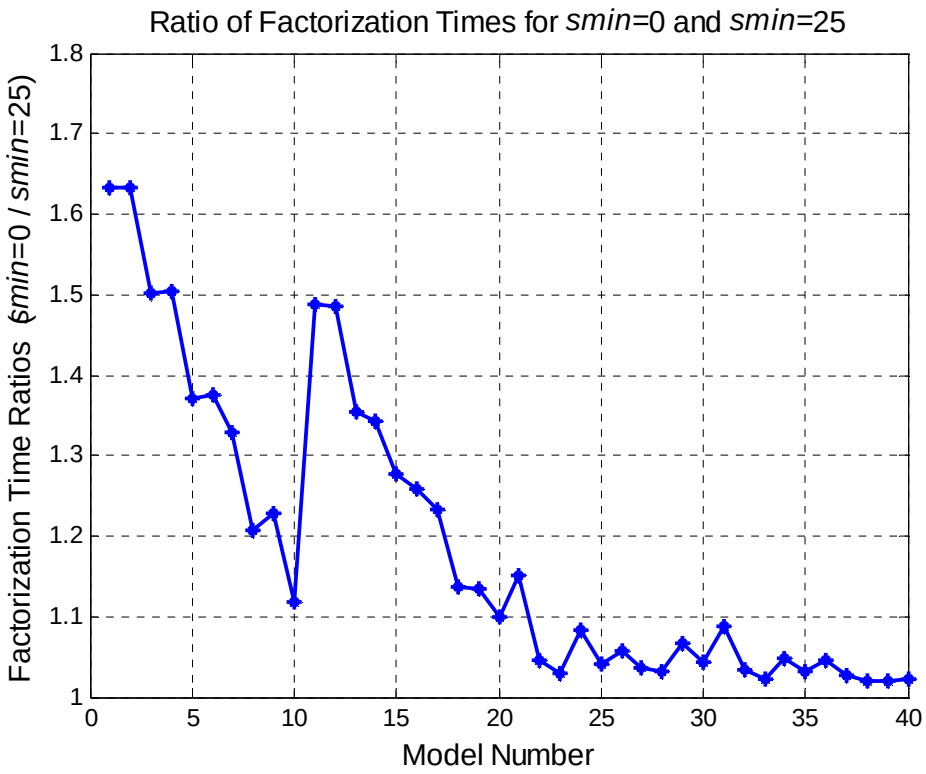


Figure 6.45: The factorization times for *smin*=0 given relative to the factorization times for *smin*=25, benchmark suite of 40 test problems.

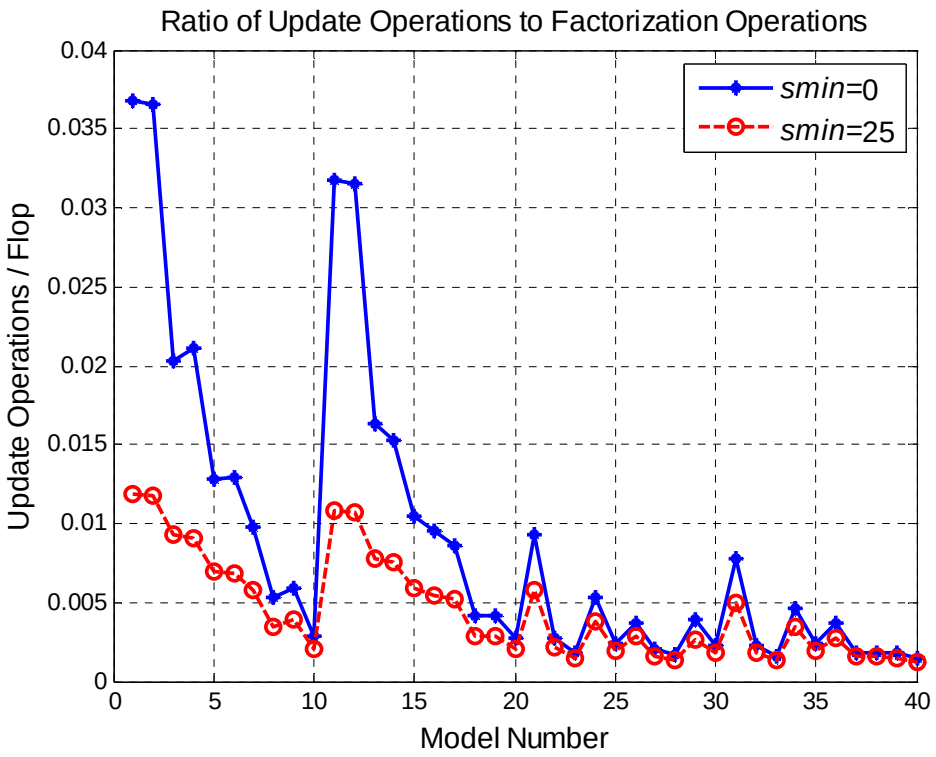


Figure 6.46: Ratio of update operations to factorization operations for *smin*=0 and *smin*=25, benchmark suite of 40 test problems.

6.5 Partitioning for Parallel Processing

As it is discussed in Chapter 3.4, explicit partitioning of the FE model may increase the factorization flop significantly. In this section, we perform numerical experiments to evaluate the increase in flop for explicit graph partitioning. Figure 6.47 shows the performance profiles for with and without explicit graph partitioning. Figure 6.47 is for benchmark suite of 40 test problems and the HMETIS ordering is used as the matrix ordering program. As shown in Figure 6.47, the application of graph partitioning typically increases the flop. There is only one exception in the benchmark suite for which explicit partitioning reduces the flop (q500×1500). Consequently, the factorization times for the explicit graph partitioning are expected to increase. Figure 6.48 shows the normalized four-thread factorization time for explicit graph partitioning and no graph partitioning. As shown in Figure 6.48, explicit graph partitioning can give best

factorization times for smaller 2D test problems with small multithreaded factorization times. The multithreaded factorization times varies significantly from a test run to another for small 2D test problems since the execution time of multithreaded BLAS kernels varies greatly for the small frontal matrices in the small test problems. In addition, the speed of multithreaded BLAS3 kernels increase as we increase the frontal matrix sizes. However, for significantly large 2D problems, explicit graph partitioning almost always increases the four-thread factorization times. As shown in Figure 6.48, explicit graph partitioning increases the multithreaded factorization times for all 3D test problems (Models 21 to 40).

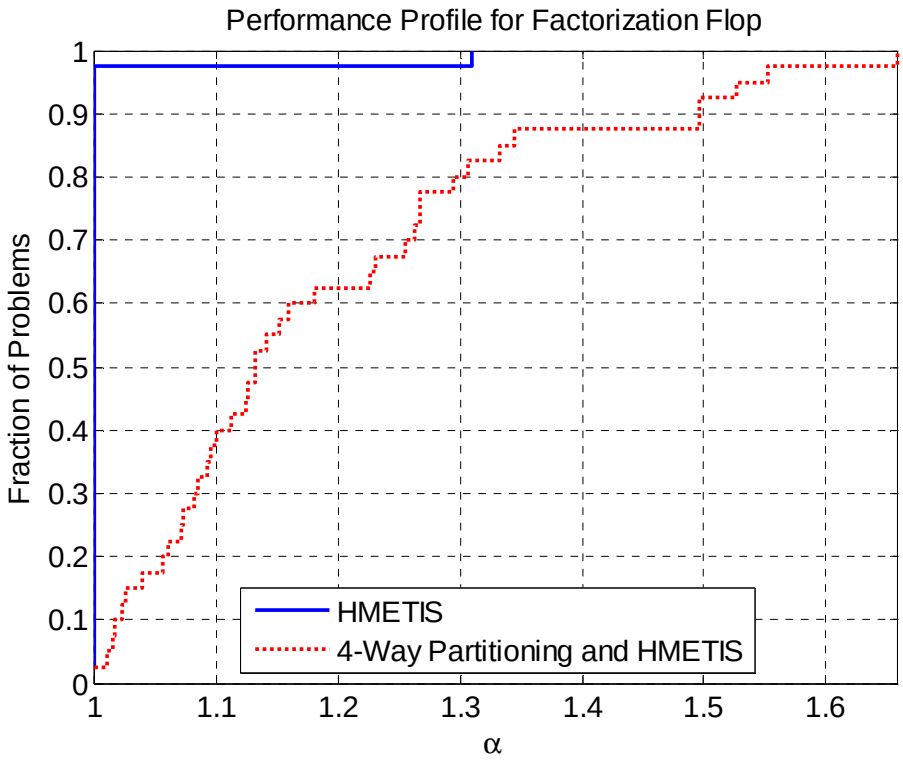


Figure 6.47: Performance profile $p(\alpha)$: Factorization flop for HMETIS ordering with and without graph partitioning.

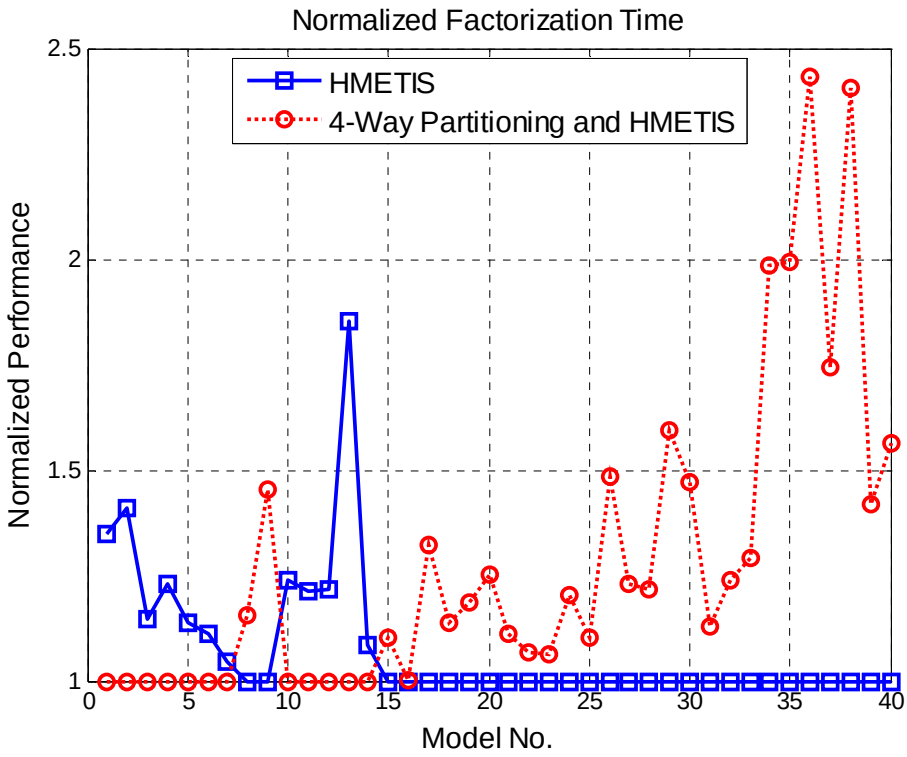


Figure 6.48: Normalized four-thread factorization time for HMETIS ordering with and without graph partitioning.

6.6 Cut-off for Node Blocking

As stated in Chapter 4.3, the execution time of node blocking can be large for tree nodes with large number of children. Typically, the low-level assembly tree nodes, the nodes close to the leaves, have large number of children. The size of the low-level assembly tree nodes is usually small. Therefore, avoiding node blocking for tree nodes smaller than a certain value, *blkmin*, will prevent executing the node blocking for nodes with large number of children. Furthermore, the size of the node blocks is small for the small frontal matrices. Performing assembly operations on small node blocks does not offer a significant performance gain. Therefore, a cut-off value that is a function of the frontal matrix size avoids applying a costly node blocking algorithm for the assembly tree nodes for which the blocked assembly is expected to offer little performance improvements.

In this section, the performance of the factorization and analysis phase is investigated for different *blkmin* values. Figure 6.49 shows the normalized factorization times for different *blkmin* values. Here, *blkmin*=0 means that node blocking algorithm is executed for all tree nodes and *blkmin*=∞ means that node blocking algorithm is never executed for any tree nodes. As shown in Figure 6.49, *blkmin*=0 usually gives the best factorization times. Figure 6.50 shows the performance profiles for the factorization times. Figure 6.50 also shows that applying node blocking to all tree nodes produces the best factorization times for most of the test problems. As shown in Figure 6.49 and Figure 6.50, turning off the node blocking (*blkmin* = ∞) may yield factorization times about 25% longer than the factorization times with node blocking for all tree nodes. Figure 6.49 shows that the node blocking reduces the factorization times for both 2D and 3D problems.

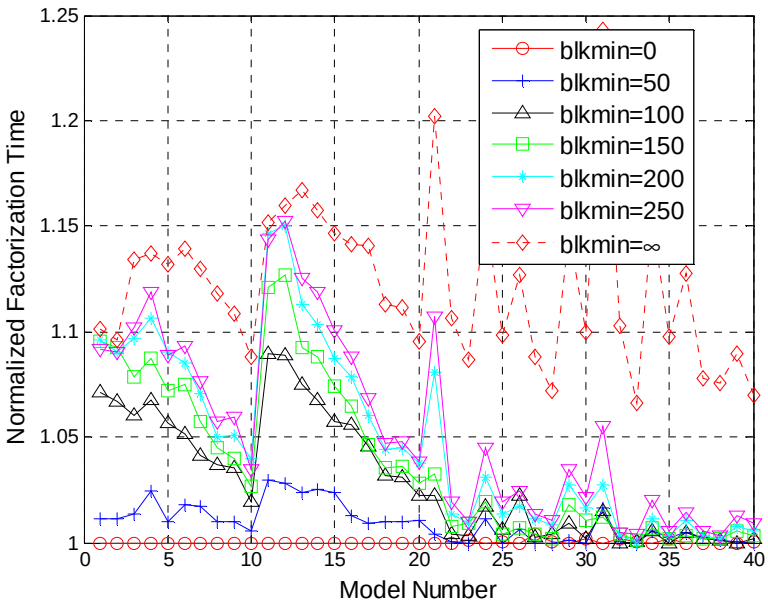


Figure 6.49: Normalized factorization time for different *blkmin* values, benchmark suite of 40 test problems.

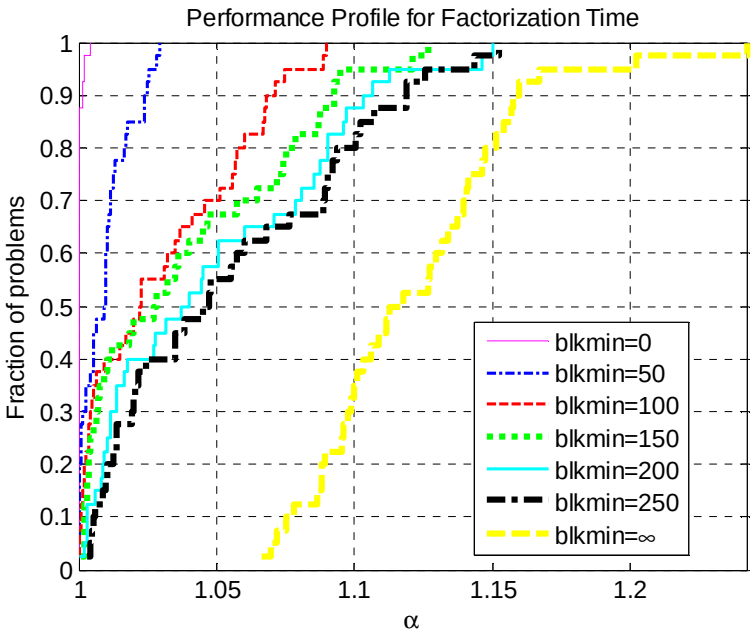


Figure 6.50: Performance profile for the factorization times for different *blkmin* values, benchmark suite of 40 test problems.

Normalized total time spent in analysis and numerical factorization phases is shown in Figure 6.51. Figure 6.51 shows that applying node blocking for all tree nodes significantly reduces the overall performance for 2D problems (models 1 to 20). This is due to the fact that execution time for analysis phase is comparable with the factorization times for 2D test problems. On the other hand, node blocking does not significantly increase the overall execution time for 3D problems. Therefore, *blkmin*=0 can be used for 3D problems. Figure 6.52 shows the performance profiles for analysis phase execution time plus factorization time. As shown in Figure 6.52, the use of *blkmin*=150 gives the best overall execution time for the current implementation of the analysis phase of the SES solver package. This value may change if we implement a more efficient analysis phase, which was not the main focus of this study. The performance profiles for using *blkmin*= 50, 100, 200, and 250 is similar to the one for using *blkmin* = 150. Therefore, *blkmin* = 50 can be used if the performance of the factorization phase is the main emphasis.

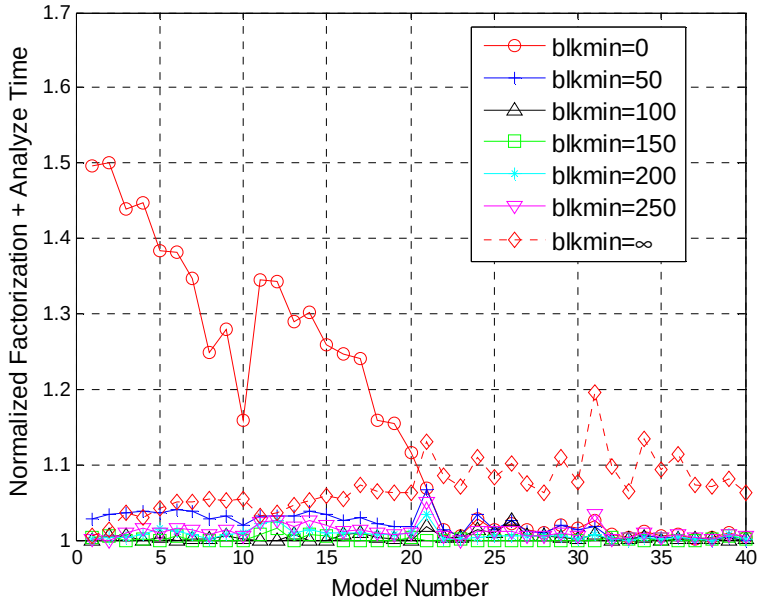


Figure 6.51: Normalized factorization plus analysis times for different *blkmin* values, benchmark suite of 40 test problems.

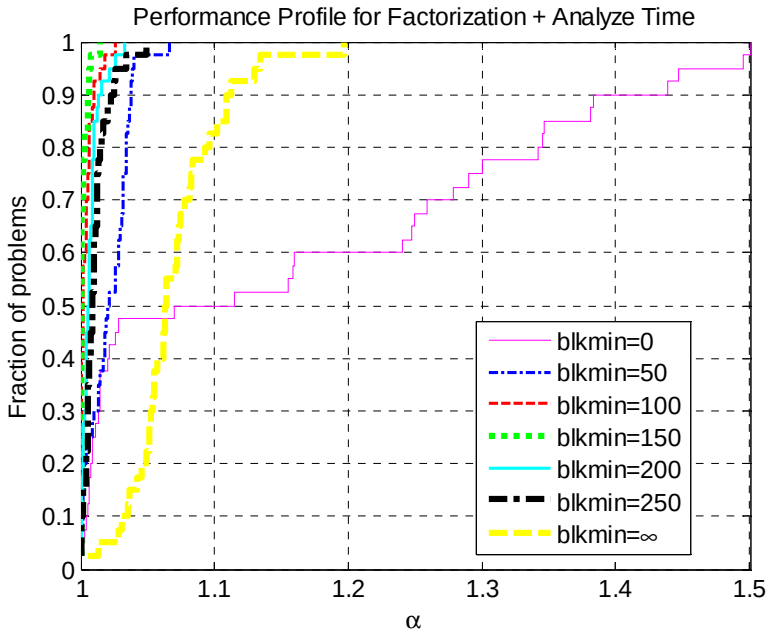


Figure 6.52: Performance profile for the factorization plus analysis times for different *blkmin* values, benchmark suite of 40 test problems.

6.7 Discussion of Results

The graph compression usually increased the flop for the pivot-orderings found by HMETIS. Consequently, the factorization times usually increased if a compressed graph is used with HMETIS. Nevertheless, the graph compression reduced the matrix ordering time. Therefore, it may reduce the overall execution time for 2D problems with significant HMETIS times relative to the factorization times. However, the local ordering AMF usually yielded the most favorable serial factorization times for 2D test problems. Moreover, for large test problems, ordering times were insignificant compared to the factorization times. Therefore, the graph compression can be avoided without any significant increase in the matrix ordering times for large 3D FE models.

The stopping criteria for nested dissections, *vertnum*, did not have a major effect on the factorization times. Therefore, the default value of *vertnum*=240, can be used for the hybrid matrix ordering program in the SCOTCH library.

For the node amalgamation algorithm discussed in Section 4.2, the amalgamation parameter *smin*=25 usually produced the most favorable factorization times. The node amalgamation improved the factorization performance of the 2D problems and small 3D problems for which ratio of number of update operations to flop is large. For the remaining 3D problems, the factorization time improvements were limited. The application of the node amalgamation within the SCOTCH library typically increased the factorization times compared to the explicit node amalgamation given in Section 4.2.

For the local matrix ordering programs, AMF, AMD, and MMD, initial numbering of the nodes based on the coordinate information proved to minimize the non-zero and flop. The improvements were significant for the matrix ordering programs AMF and MMD. For the hybrid matrix ordering programs, there was no single initial node numbering that consistently yields favorable non-zero or flop. Nevertheless, the non-zero and flop can be minimized by trying alternative random node permutations for the hybrid matrix ordering programs.

Compared to five matrix ordering programs, AMF usually yielded favorable factorization times for 2D test problems. It also yielded favorable factorization times for 3D test problems with 2D-like geometries and 3D test problems for which nodes are connected to a small number of adjacent nodes. For the remaining 3D test problems, HMETIS usually yielded favorable factorization times. Furthermore, HMETIS usually gave the pivot orderings at half time of other hybrid matrix ordering program, HAMF.

The time spent in the hybrid matrix ordering programs was comparable to the factorization times for smaller 2D test problems. On the other hand, the factorization time usually dominated the total execution time for large 2D FE models and for most of the 3D FE models. Therefore, it is crucial to minimize the factorization time for large 2D FE models and for 3D models. To minimize the factorization times, alternative matrix ordering strategies can be executed and the pivot-ordering yielding the best estimated factorization time can be used for the factorization and triangular solution.

Numerical experiments showed that the ratio of the analysis time to the factorization time decreases as the number of dofs in a FE model increases. For 3D FE models having more than 30,000 dofs, the analysis phase took less than 10% of the factorization time for the PARDISO solver.

The coarsening scheme proved to reduce the matrix ordering times. Furthermore, using a coarser scheme in the matrix ordering programs yielded favorable factorization times for the majority of the test problems. The reduction in factorization times were illustrated for a scheme that chooses the pivot-ordering among the results of the matrix ordering programs executed for the original and coarsened meshes.

It was shown that performing the node blocking selectively reduces the analysis time but increases the factorization time. The node blocking cut-off value of 50 (*blkmin*=50) gave satisfactory overall execution time without a significant decrease in the factorization times.

CHAPTER 7

SOLVER PERFORMANCE

Numerical experiments are performed to demonstrate the performance of the SES solver package. The performance of the SES solver package is compared with the PARDISO solver package. PARDISO is a high performance sparse direct solver for shared memory processors. Gould et al. [111] illustrated PARDISO's efficiency for serial direct solution of a symmetric system of equations. We compare the numerical factorization times with the estimated factorization times and PARDISO factorization times. The efficiency of the triangular solution is also illustrated. Finally, the performance of an out-of-core version of the solver is demonstrated for eight very large test problems.

7.1 In-core Solver

In the multifrontal method, the factors can be written to the disk as soon as they are computed, which reduces the memory footprint for the numerical factorization and triangular solution. However, this typically degrades the performance. The performance of the SES solver is first evaluated keeping the factors in the main memory.

7.1.1 Serial Solver

The single thread performance of the multifrontal solver is evaluated for the benchmark suite of 40 test problems. Figure 7.1 shows the factorization speed for the PARDISO and SES solver packages. In order to make this comparison as fair as possible, HMETIS ordering is used for both solver packages which is the default ordering in PARDISO. The SES solver is provided with a pivot-ordering found by HMETIS ordering with the graph compression. As shown in Figure 7.1, the factorization speeds of the SES and PARDISO solvers greatly depend on the test problem. The factorization speed is

usually high for large problems. The frontal matrices are large for large problems and the BLAS3 kernels run faster for larger frontal matrices as shown in Section 4.4.1. The factorization speeds are also influenced by the number of operations required for the update matrices per each arithmetic operation required for the factorization. As this ratio increases, the speed of factorization decreases since the update matrix copy and assembly operations performed at slower speeds compared to the partial factorization. This is mainly due to the low speeds of memory copy operations relative to the BLAS3 speeds for partial factorization. The ratios of update matrix operations to factorization operations have been previously shown in Figure 6.46 in the previous Chapter. The problems with a large ratio of update matrix operations to factorization operations in Figure 6.46 usually have smaller factorization speeds in Figure 7.1.

As shown in Figure 7.1, the factorization speeds for PARDISO and SES are similar even though two solver packages implement different factorization schemes. Namely, PARDISO implements a mixture of left and right looking schemes while SES implements a multifrontal method. The use of a different factorization scheme does not make a significant difference on the speeds of the factorization. This finding is similar to the findings of Gould et al. [111]. Figure 7.1 also shows the upper bound for the speed of multifrontal factorization. The upper bound is found by executing the BLAS3 kernels for the frontal matrices corresponding to the assembly trees built for the test problems. The overhead of handling sparse data structures (such as assembly of update matrices) and assembly of the FE matrices are neglected for finding the upper bounds. As shown in Figure 7.1, the factorization speeds qualitatively follow the upper bounds. Therefore, the low factorization speeds for smaller test problems is partly due to the low speeds of BLAS3 kernels on small frontal matrices. The factorization speed approaches the upper bound for large problems where the ratio of factorization operations to memory accesses increases.

As the number of RHS vectors increases, the triangular solution speed also increases due to the increase in the size of the frontal matrices for which the BLAS3 operations are performed. Figure 7.2 shows the speed of the solution phase for 100 RHS vectors. Compared to the factorization speed, the speed variations between the test problems are smaller for the solution with 100 RHS vectors. Figure 7.2 also shows that the back substitution is slightly faster than the forward elimination. The relative performance of forward elimination and back substitution is similar for the PARDISO solver package. Figure 7.2 further shows the upper bound for the speed of forward elimination and back substitution. The performance difference between forward elimination and back substitution is also visible in the upper bound plots of the triangular solution speeds for the 3D problems (Models 21 to 40).

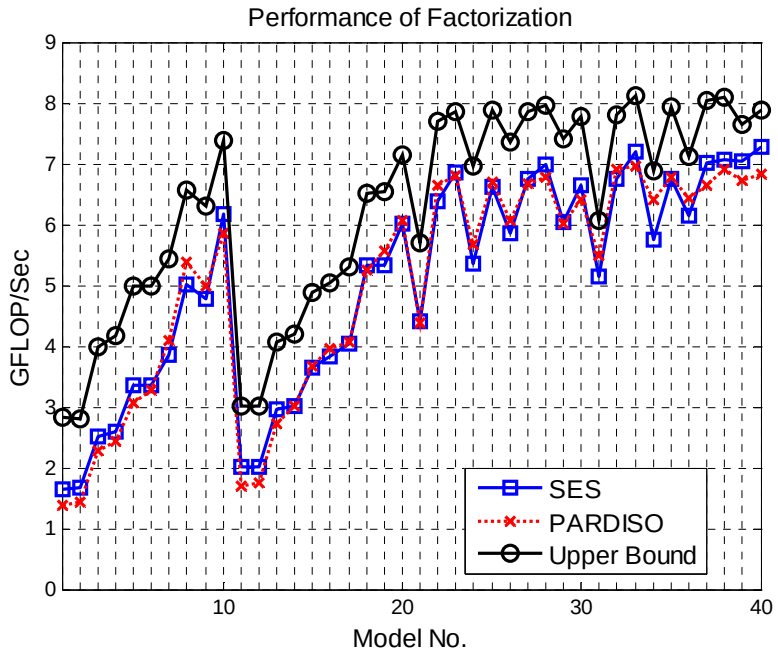


Figure 7.1: Speed of single thread factorization for SES and PARDISO solver. Pivot-orderings are found with HMETIS. Benchmark suite of 40 test problems.

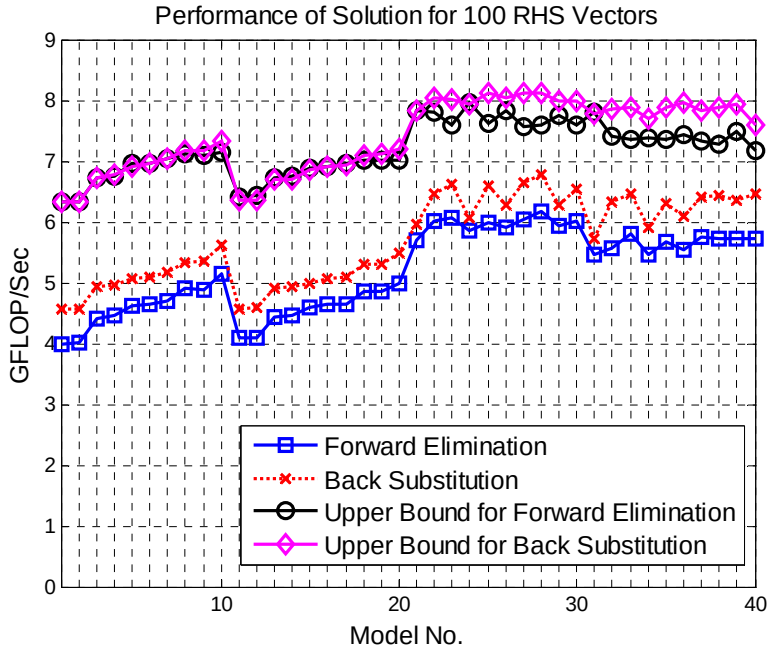


Figure 7.2: For SES solver package, speed of single thread solution for 100 RHS vectors. Pivot-orderings are found with HMETIS. Benchmark suite of 40 test problems.

Figure 7.3 shows the performance profiles for the serial factorization times with SES and PARDISO solver packages. As shown in Figure 7.3, SES gives the best factorization times for 80% of the test problems. The factorization times for SES solver are within the 1.2 times the factorization times with the PARDISO package except from one problem out of 40 test problems in the benchmark suite. PARDISO gives a significantly better factorization time for the test problem f30×30×30 (Model No. 38). For this problem, the PARDISO factorization flop is significantly smaller than the SES factorization flop. The number of arithmetic operations required for factorization (flop) mainly determines the factorization times since the speed of factorization is similar for both solvers as shown in Figure 7.2. Namely, PARDISO requires 383.41 GFlop for numerical factorization while SES requires 540.23 GFlop using HMETIS with graph compression and 438.76 GFlop using HMETIS without graph compression.

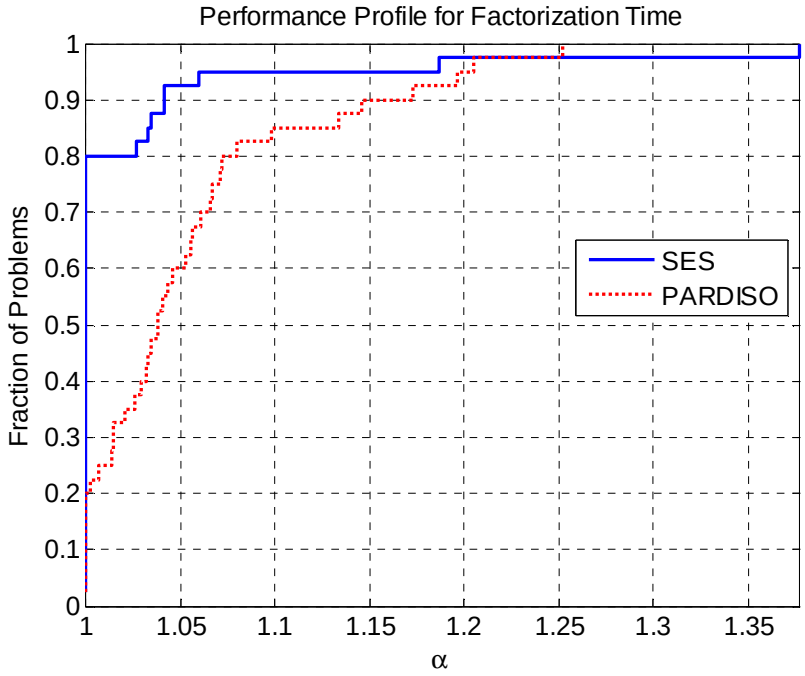


Figure 7.3: Performance profile for serial factorization times. Pivot-ordering is found with HMETIS. Benchmark suite of 40 test problems.

The comparison of the factorization times is not a fair performance assessment since SES interleaves the stiffness matrix assembly operations with the numerical factorization steps. Therefore, the factorization times already include the time required for the assembly of the FE stiffness matrices for the SES solver package. For a fair comparison, the overall time required for stiffness matrix assembly and factorization is used to evaluate the performance of the numerical factorization phases of the solver packages. Figure 7.4 shows the performance of the numerical factorization plus assembly times for the serial execution of the solvers. As shown in Figure 7.4, the performance of SES is significantly better than the performance of the PARDISO when the time required for the assembly is considered.

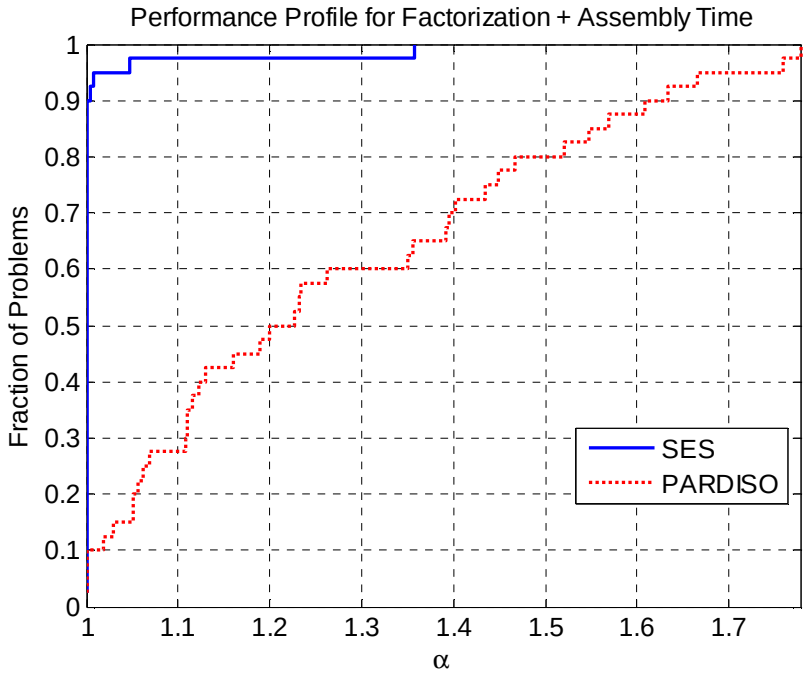


Figure 7.4: Performance profile for serial factorization plus assembly times. Pivot-ordering is found with HMETIS. Benchmark suite of 40 test problems.

Figure 7.5 shows the performance profile for the serial solution with 100 RHS vectors. As shown in Figure 7.5, SES consistently outperforms the PARDISO solver in

the triangular solution phase. PARDISO gives triangular solution times that are larger than 2 times of the SES triangular solution times for about 50% of the benchmark problems.

For selected test problems, the execution times of the SES and PARDISO solver are given in Table 7.1 and Table 7.2 respectively. The execution time of matrix ordering program and analysis phase are also shown in the tables.

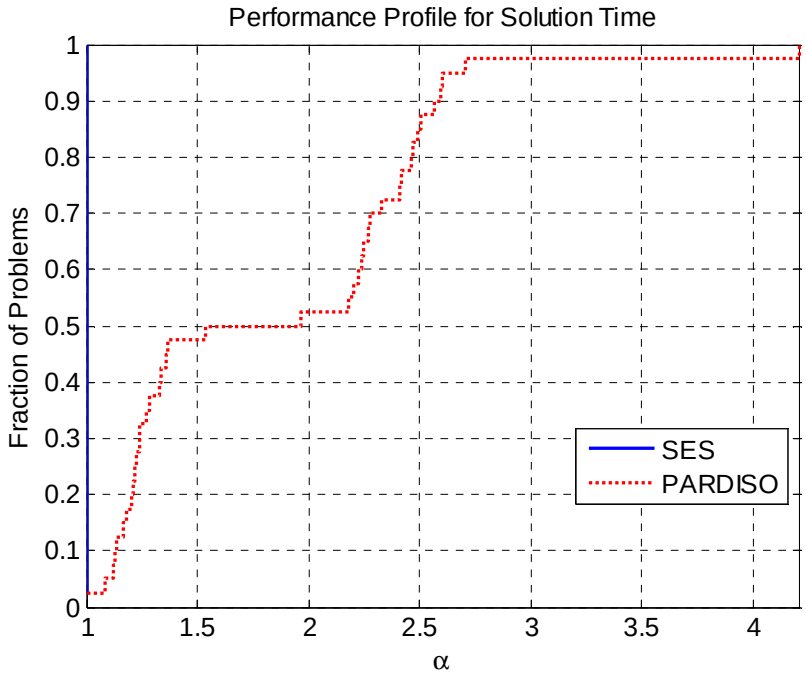


Figure 7.5: Performance profile for serial triangular solution times for 100 RHS vectors. Pivot-ordering is found with HMETIS.

Model Name	Matrix-Ordering Time (sec)	Analysis Time (sec)	Factorization Time (sec)	Triangular Solution Time (sec)	Total Time (sec)
q500×500	1.7	0.8	4.7	4.6	11.8
f500×500	1.3	1.2	9.0	6.6	18.1
s30×30×30	0.3	0.3	20.0	4.7	25.3
f30×30×30	0.2	0.3	76.6	10.4	87.5

Table 7.1: Serial execution time of the different phases of the SES solver

Model Name	Matrix-Ordering Time (sec)	Analysis Time (sec)	Factorization Time (sec)	Triangular Solution Time (sec)	Total Time (sec)
q500×500	1.8	1.1	5.5	10.0	18.4
f500×500	2.1	1.4	8.7	14.9	27.1
s30×30×30	0.6	0.4	21.6	5.7	28.3
f30×30×30	0.4	0.6	55.6	10.6	67.2

Table 7.2: Serial execution time of the different phases of the PARDISO solver

Finally, we evaluate the serial in-core performance of the SES solver package for 8 large test problems given in Chapter 2. For the SES solver package, the AMF matrix ordering is used for 2D test problems. As illustrated in Chapter 6, the AMF ordering usually produces pivot orderings with favorable non-zero and flop for 2D test problems. Furthermore, its execution time is significantly smaller than the HMETIS matrix ordering program. For 3D test problems, the HMETIS matrix ordering is used. Graph compression is not applied for HMETIS since this may increase the flop and non-zero for an input model (see Chapter 6). For the SES solver package, a strategy is used which minimizes the factorization time by selecting the best pivot-ordering among several alternatives. In this strategy, node based coarsening is applied to the original FE meshes for the *nodeco*=1 and 2 values. The pivot-ordering that yields the best estimated factorization time is chosen among the results of the matrix ordering programs for the original and coarsened meshes. This pivot ordering is used for numerical factorization and triangular solution. Table 7.3 shows the configurations that produced the pivot-ordering with best estimated factorization time for these test problems. As shown in Table 7.3, matrix ordering programs executed for the coarsened meshes yielded the best estimated factorization times for half of the large test problems. For 2D models (first four models in Table 7.3), the AMF ordering yielded estimated factorization times better than the HMETIS counterpart.

Model No./Name	Preprocessing Configuration	Estimated Factorization Time
1.Q2DL1	*nodeco*=1 + AMF	15.79
2.Q2DL2	AMF	19.14
3.F2DL1	*nodeco*=2 + AMF	20.36
4.F2DL2	*nodeco*=1 + AMF	21.94
5.S3DL1	HMETIS	190.21
6.S3DL2	*nodeco*=1 + HMETIS	151.45
7.F3DL1	HMETIS	170.59
8.F3DL2	HMETIS	233.92

Table 7.3: Preprocessing configuration that produces the best estimated serial factorization times for the benchmark suite of 8 large test problems.

Figure 7.6 shows the factorization times for the 8 large test problems normalized according to the 'PARDISO factorization plus stiffness matrix assembly' times. Figure 7.6 also shows the PARDISO factorization times without including the time required for the assembly of the stiffness matrix. As stated previously, the SES solver package interleaves stiffness matrix assembly operations with the numerical factorization operations. Therefore, factorization times for SES solver package already include stiffness matrix assembly times. As shown in Figure 7.6, factorization with SES outperforms factorization with PARDISO for 7 out of 8 large test problems even if the assembly time is not included with the PARDISO factorization times. If the assembly times are also included, SES outperforms PARDISO for all test problems. Using the factorization time minimization strategy, SES can be 1.75 times faster than the PARDISO solver package. The speedup of SES is mainly due to smaller flops in the preprocessing phase of the SES solver. As stated previously, both solvers run at similar speeds at the numerical factorization phase (excluding the speed of assembly operations for PARDISO). Therefore, relative serial factorization performance is greatly determined by the number of floating point operations required for factorization.

Figure 7.6 also shows the estimated factorization times for the SES solver package. As shown in Figure 7.6, the estimated factorization times closely follows the

actual factorization times. Therefore, the performance of the numerical factorization meets the expectations for single thread factorization of this set of large test problems.

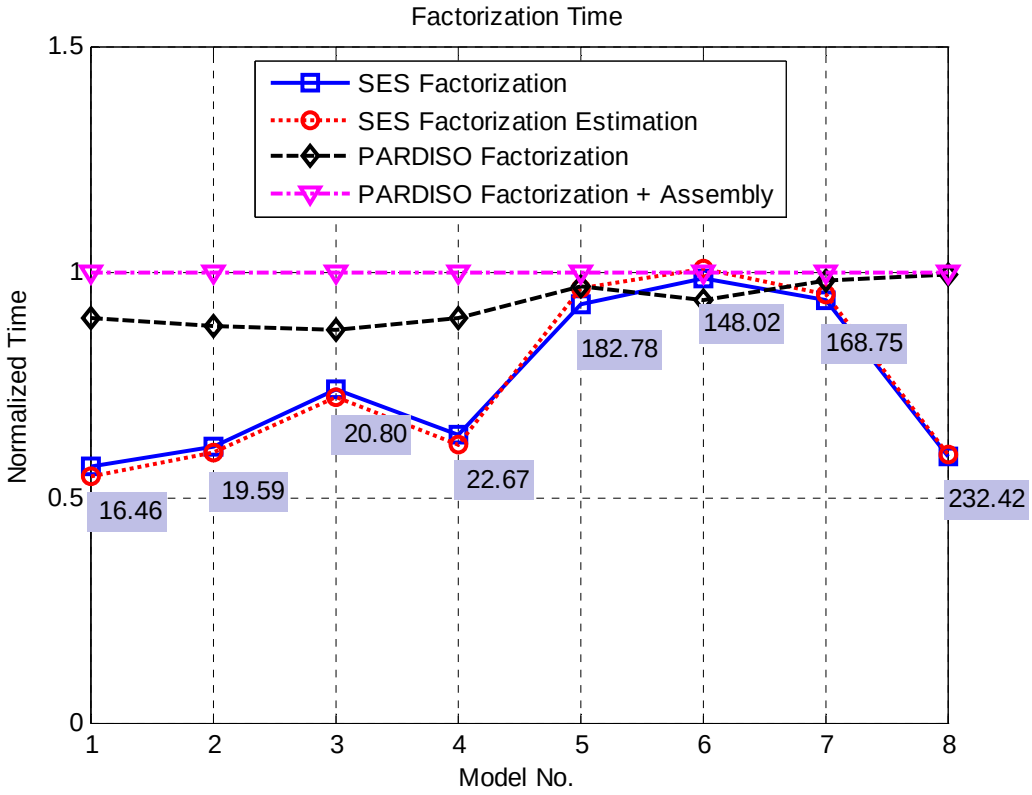


Figure 7.6: Serial numerical factorization times normalized according to the PARDISO numerical factorization plus assembly times for 8 large test problems. SES factorization times (in seconds) are also shown in the blue boxes.

Figure 7.7 shows the triangular solution times with 100 RHS vectors normalized according to the PARDISO triangular solution times. As shown in Figure 7.7, SES solver package consistently outperforms PARDISO solver. SES solver package is especially efficient for the triangular solution of 2D test problems. For these problems, the preprocessing strategy employed for the SES solver reduces the floating point operations required for the solution. SES performs triangular solution 2.55 times faster than PARDISO for Model 4 (F2DL2) as shown in Figure 7.7. If we compare the factorization times with triangular solution times shown in Figure 7.6 and 7.7 respectively, we observe

that the triangular solution with 100 RHS vectors has comparable execution times with respect to the numerical factorization times for 2D models. Therefore, it is especially important to improve the performance of triangular solution with a large number of RHS vectors for 2D test models. As shown in Figure 7.7, the SES solver package performs the triangular solution of the 2D models significantly faster than PARDISO (the speedup is larger than 2 for all 2D test problems).

It should be noted that triangular solution phase of the SES solver package is tuned for the solution of large number of RHS vectors. For a small number of RHS vectors, the triangular solution with PARDISO usually outperforms the SES solver package (see Section 5.2.2. for details).

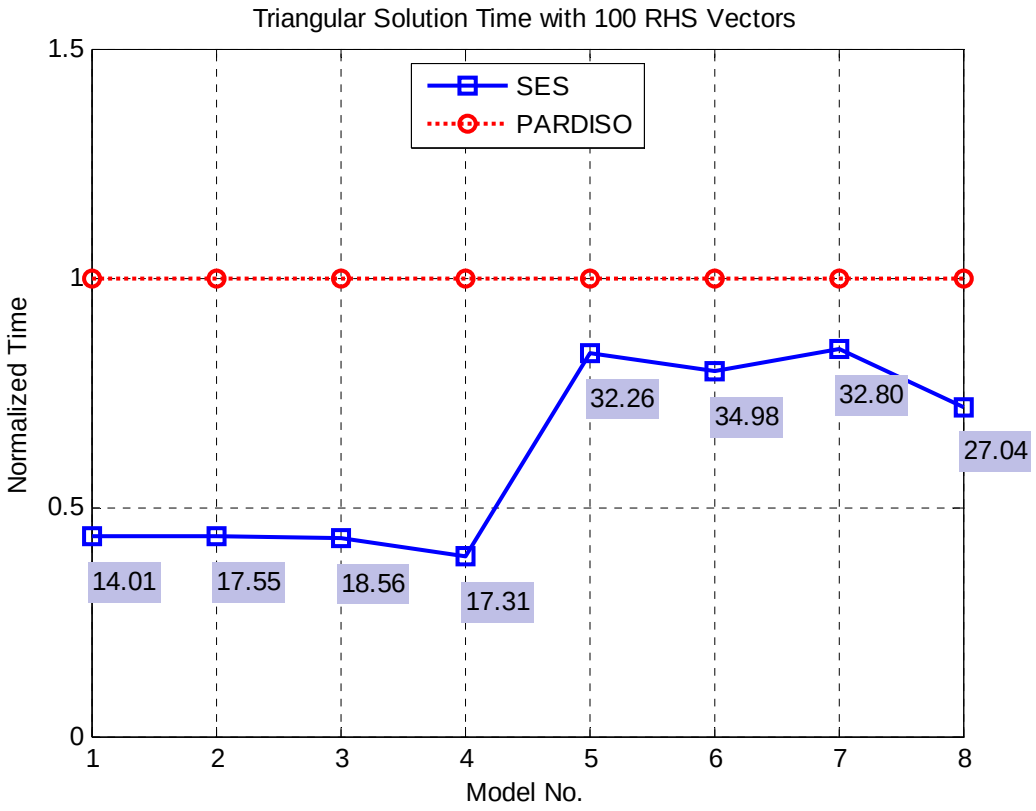


Figure 7.7: Serial triangular solution times normalized according to the PARDISO triangular solution times for 8 large test problems. SES triangular solution times (in seconds) are also shown in blue boxes.

7.1.2 Multithreaded Solver

In this section, the performance of four-thread factorization and triangular solution is presented for the SES solver package. Figure 7.8 shows the speed of four-thread factorization for the SES and PARDISO solver packages for benchmark suite of 40 test problems. As shown in Figure 7.8, the four-thread factorization speed for SES is better than PARDISO except from Model 34. Figure 7.9 shows the speed of triangular solution with 100 RHS vectors using four threads. As shown in Figure 7.8 and Figure 7.9 the four-thread factorization and triangular solution speeds typically increase as the size of the problems increase.

Next, we compare the four-thread execution times of the SES and PARDISO solver packages. For SES and PARDISO, the performance profiles for the four-thread factorization times are shown in Figure 7.10. The factorization times for PARDISO do not include the time required for the assembly of the stiffness matrix. SES gives significantly better four-thread factorization times compared to PARDISO even though the stiffness matrix assembly times are not included for PARDISO factorization. Figure 7.11 shows the factorization plus assembly times for the two solver packages. As shown in Figure 7.11, PARDISO factorization times are more than 2 times the SES factorization times for about 40% of the test problems. Figure 7.12 shows performance profile for the solution times for 100 RHS vectors. As shown in Figure 7.12, SES generally outperforms PARDISO for the triangular solution with 100 RHS vectors. PARDISO triangular solution times are larger than 2 times the SES counterparts for about 50% of the problems in the benchmark suite. For a single test problem (f500×1500), the solution time for SES is about 30 times faster than the one for PARDISO. This is due to the excessive paging of PARDISO for the solution of this particular problem.

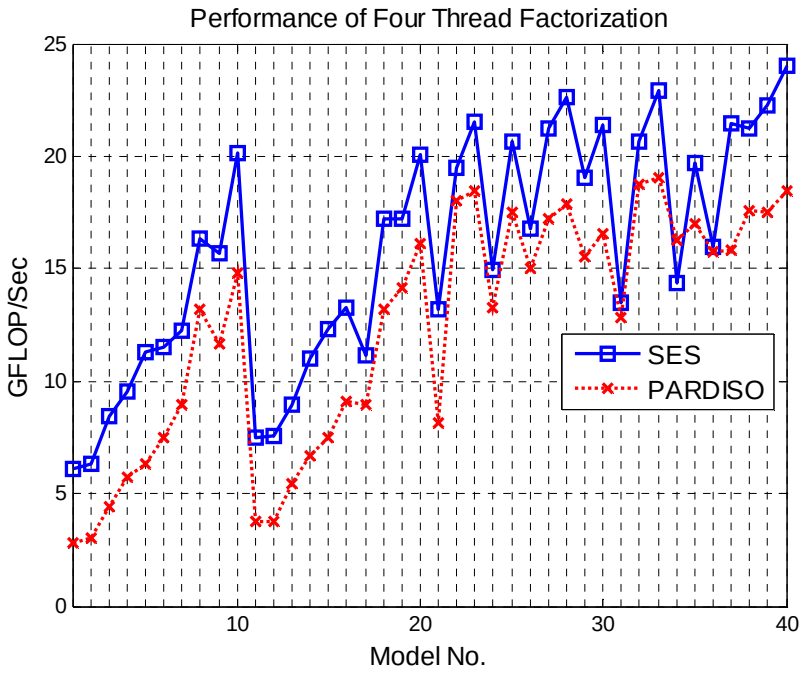


Figure 7.8: Speed of four-thread factorization for SES and PARDISO solver.

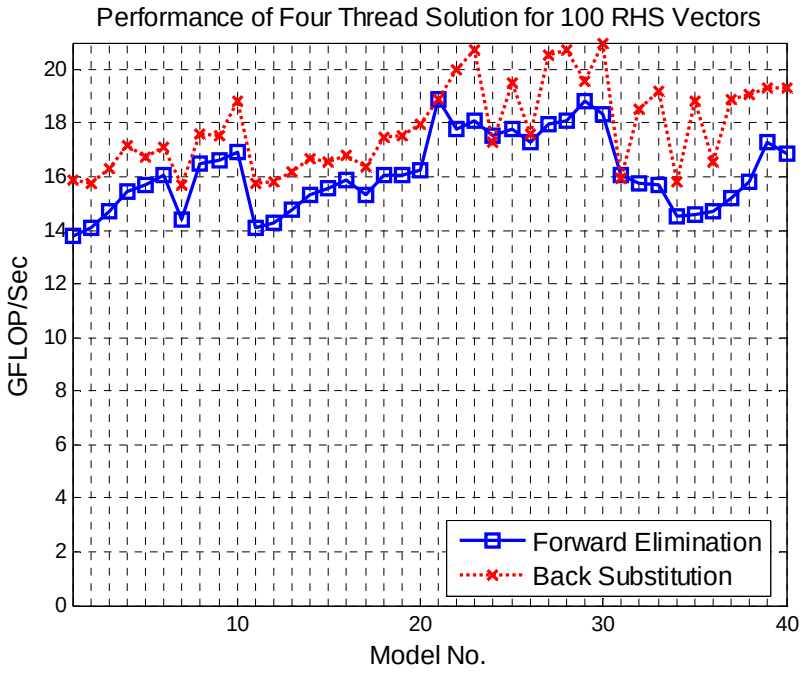


Figure 7.9: For SES solver, speed of four-thread solution with 100 RHS vectors

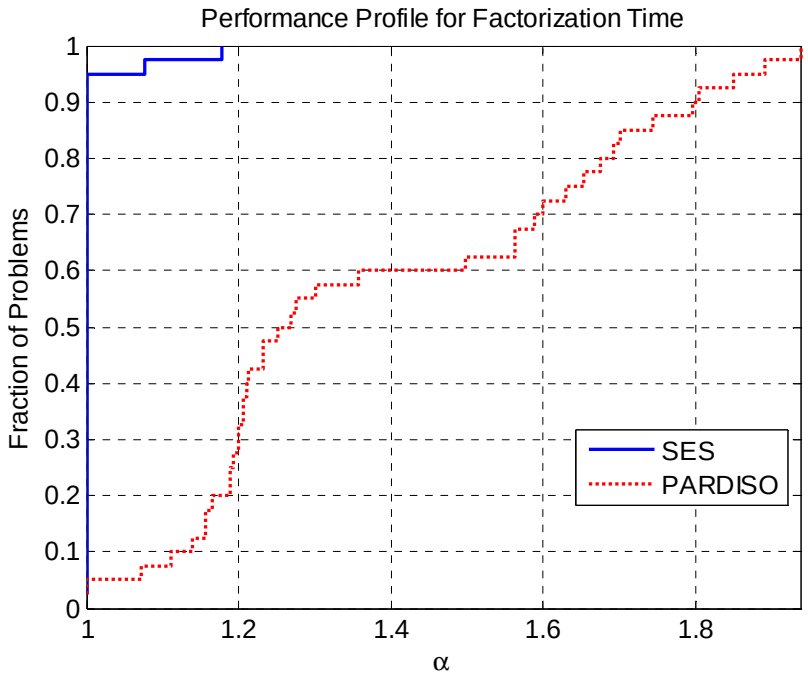


Figure 7.10: Performance profile for four thread factorization times for SES and PARDISO solver

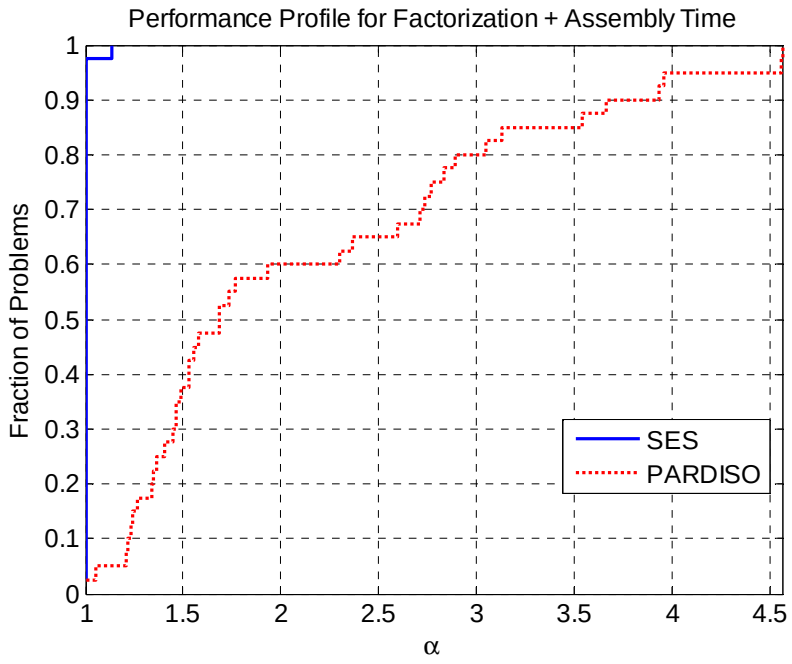


Figure 7.11: Performance profile for four thread factorization plus assembly times for the SES and PARDISO solvers.

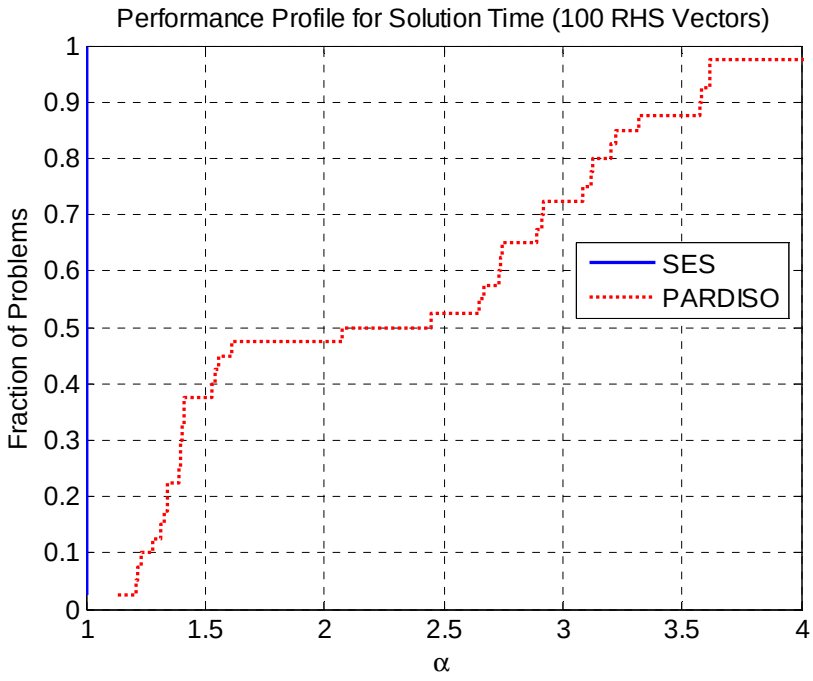


Figure 7.12: Performance profile for four-thread solution times for the SES and PARDISO solvers.

Next, we demonstrate the four-thread numerical factorization and triangular solution performance of the SES solver package on 8 large test problems described in Chapter 2. We compare the multithreaded performance of the SES solver with the PARDISO solver. For the SES solver package, we find alternative pivot-orderings for different preprocessing configurations and choose the pivot-ordering that yields the best estimated multithreaded factorization time. Table 7.4 shows the configuration that yields the best estimated four-thread factorization times for the benchmark suite of 8 large test problems. As shown in Table 7.4, AMF gives the best estimated factorization time for 3 out of 4 2D test problems. HMETIS yields better four-thread factorization time for the test problem F2DL2 for which the AMF was superior for serial factorization as it was shown in Table 7.3.

Figure 7.13 shows the four-thread factorization times normalized according to 'PARDISO factorization plus assembly' times. As shown in Figure 7.13, SES

consistently outperforms PARDISO even though the assembly times are not included for the PARDISO solver. If we include the assembly times for the PARDISO, SES is more than 2 times faster than PARDISO for half of the large test problems as shown in Figure 7.13. For the PARDISO solver, the percentage of assembly times increases for the four-thread factorization compared to the serial factorization since the assembly operations are performed in a serial fashion. Figure 7.13 also shows the estimated four-thread factorization times for the SES solver package. As shown in Figure 7.13, the numerical factorization is slower than what it is predicted to be. We discuss the difference between the estimated and actual performance in the following section.

Figure 7.14 shows the four-thread triangular solution times normalized according to the PARDISO triangular solution times. As shown in Figure 7.14, SES generally outperforms PARDISO for the triangular solution with 100 RHS vectors. SES triangular solution is 3.2 times faster than PARDISO triangular solution for the test problem Q2DL2.

Model No./Name	Preprocessing Configuration	Estimated Factorization Time (seconds)
1.Q2DL1	*nodeco*=1 + AMF	4.37
2.Q2DL2	*nodeco*=1 + AMF	5.20
3.F2DL1	HMETIS	5.93
4.F2DL2	*nodeco*=1 + AMF	5.92
5.S3DL1	HMETIS	49.87
6.S3DL2	HMETIS	39.76
7.F3DL1	*nodeco*=1 + HMETIS	45.95
8.F3DL2	*nodeco*=1 + HMETIS	63.96

Table 7.4: Preprocessing configuration that produces the best estimated four-thread factorization times for the benchmark suite with 8 large test problems.

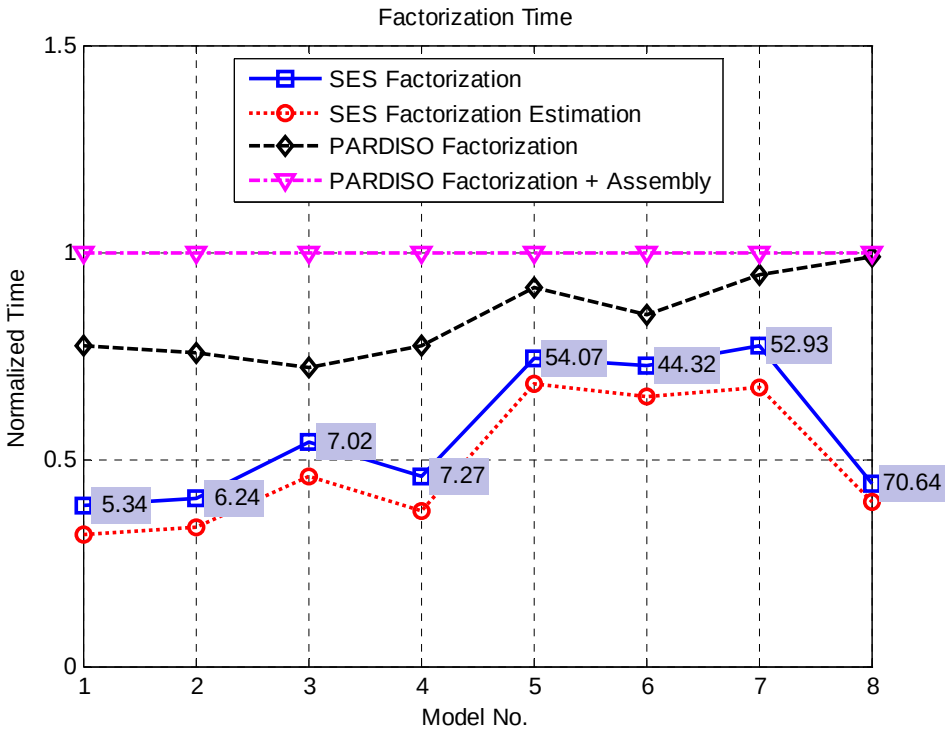


Figure 7.13: Four-thread numerical factorization times normalized according to the PARDISO numerical factorization plus assembly times for 8 large test problems. SES factorization times (in seconds) are also shown in the blue boxes.

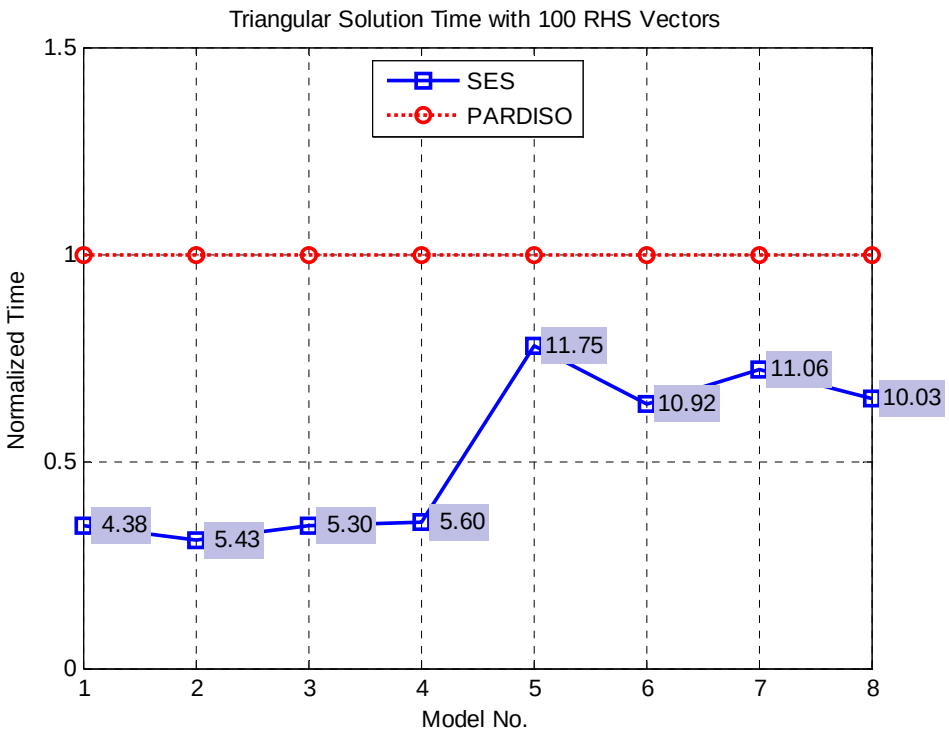


Figure 7.14: Four-thread triangular solution times normalized according to the PARDISO numerical factorization plus assembly times for 8 large test problems. SES triangular times (in seconds) are also shown in the blue boxes.

Both PARDISO and SES solver packages require more memory for a four-thread solution compared to the single thread solution. The increase in memory requirement is less significant for the PARDISO solver package compared to SES. The number of active frontal matrices and update matrix stacks are as many as the number of threads assigned to the independent subtrees for the SES solver package. The use of multiple frontal matrices and update matrix stacks increases the active memory requirement of the SES numerical factorization phase. Table 7.5 shows the memory requirements for factors, frontal matrices and update matrix stacks of the SES solver package. Table 7.5 gives the memory requirements for serial and four-thread numerical factorization. As shown in Table 7.5, the increase in memory is significant for 3D test problems. For the test

problem F3DL2, four-thread factorization requires 1.33 times the memory required for the serial factorization.

Model No./Name	Memory (Gbytes) for Serial Factorization	Memory (Gbytes) for Four-thread Factorization	Increase in Memory Requirement
1.Q2DL1	1.98	2.18	×1.10
2.Q2DL2	2.23	2.42	×1.09
3.F2DL1	2.24	2.35	×1.05
4.F2DL2	2.42	2.73	×1.13
5.S3DL1	5.68	6.67	×1.17
6.S3DL2	5.75	6.54	×1.14
7.F3DL1	5.47	6.78	×1.24
8.F3DL2	4.99	6.65	×1.33

Table 7.5: Serial and multithreaded memory requirements of the SES factorization.

7.1.3 Analysis of the Multithreaded Performance

As shown in Figure 7.13, SES four-thread numerical factorization takes slightly longer than the estimated numerical factorization time. The reasons for this unanticipated performance degradation are investigated. The estimated and actual factorization times are compared for the subtree and high-level assembly tree nodes. As described previously, the subtree tree nodes are processed in parallel by employing single-threaded BLAS3 kernels. On the other hand, the high-level tree nodes are processed by employing multi-threaded BLAS3 kernels within the main thread. Figure 7.15 shows the subtree factorization times and estimated subtree factorization times for the benchmark suite with 8 large test problems. The subtree factorization times given in Figure 7.15 are normalized according to the overall time required for the factorization. As shown in Figure 7.15, the subtree factorization times may take between 64%-96% of the total factorization time for the four-thread factorization of the test problems. As shown in Figure 7.15, the subtree factorization executes slower than the expected performance. In the worst case, the actual subtree factorization time is 36% larger than the estimated subtree factorization time

(Model 8). The factorization of the subtrees is performed simultaneously by executing single-threaded BLAS3 kernels. We simulate the subtree partial factorization operations by executing serial BLAS3 kernels simultaneously by different threads. The experiments with BLAS3 kernels show that the performance is degraded if multiple threads execute BLAS3 kernels independently. The performance is degraded even if the number of threads executing serial BLAS3 kernels is smaller than the number of physical cores in the system. As an example, in our test system with four cores, we observe about 10% increase in the partial factorization time for the simultaneous partial factorization of three frontal matrices with 500 eliminated and 1000 remaining variables. The performance degradation of BLAS3 kernels may be the due to the memory bus and cache contention of the threads. Consequently, we conjecture that the computational resource contention due to simultaneous execution of BLAS3 kernels contribute to the performance degradation of the numerical factorization. Presumably, the resource contention also slows down the speed of the update matrix copy and assembly operations. The current performance model does not consider the performance degradation due to the computational resource contentions during the simultaneous factorization operations on the subtrees. Hence, the underestimation of the subtree factorization times is the main reason for the under prediction of the overall execution times.

Figure 7.16 shows the actual and estimated high-level tree node factorization times normalized according to the total factorization time. As shown in Figure 7.16, the high-level factorization times are significantly overestimated for Models 3 and 8 since the speedup model for the BLAS3 kernels is simply a function of partial factorization operation counts. Furthermore, the performance model does not consider the speedup of update matrix assembly operations. Even though the high-level factorization times are predicted accurately or overestimated, the total factorization times are underestimated due to the performance degradation of the subtree factorization explained previously.

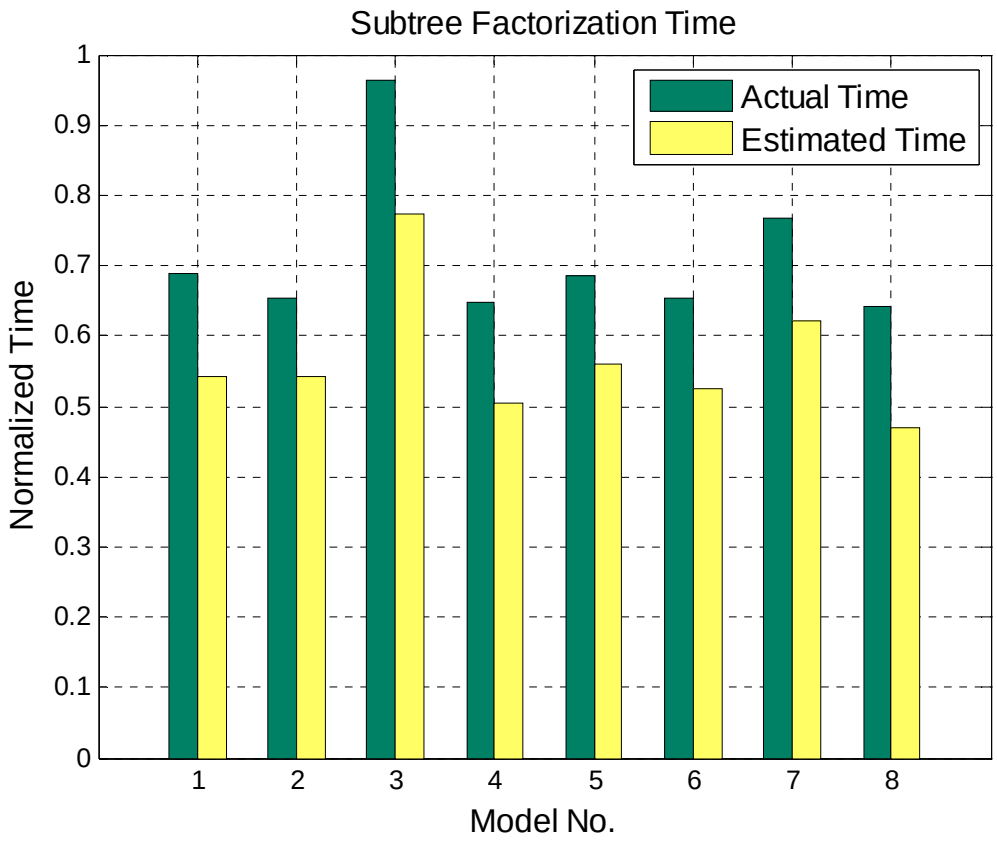


Figure 7.15: Estimated and actual subtree factorization times normalized according to the total numerical factorization time for the benchmark suite with 8 large test problems (HMETIS)

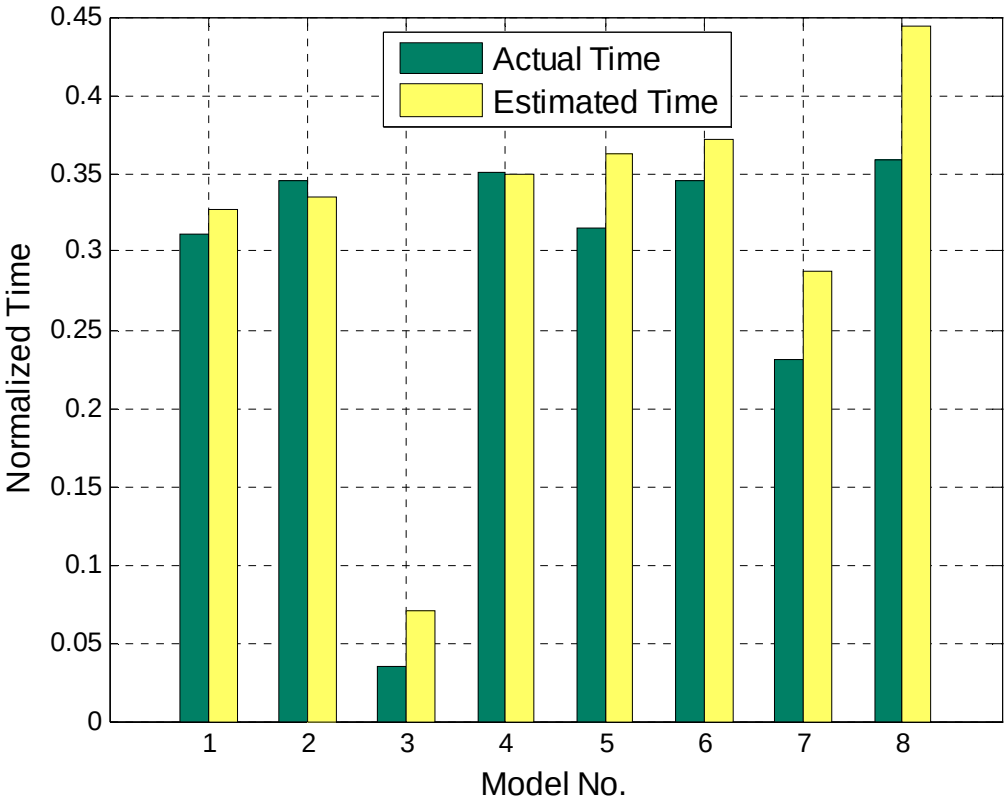


Figure 7.16: Estimated and actual high-level tree node factorization times normalized according to the total numerical factorization time for the benchmark suite with 8 large test problems (HMETIS)

For the benchmark suite with 8 large test problems, the SES speedups for four thread numerical factorization and triangular solution are given in Figure 7.17. For four-thread factorization, the SES numerical factorization speedup is around 3.3 for all 8 large test problems as shown in Figure 7.17. Unlike speedups for the numerical factorization, the speedups for forward elimination and back substitution vary significantly for the large test problems. In the triangular solution phase, we use the mapping found for the numerical factorization. Therefore, the variation in speedups is expected since the workload between the threads is not necessarily balanced for the triangular solution.

Figure 7.18 shows the speedups for the PARDISO solver package executed with four threads for the benchmark suite with 8 large test problems. If we compare the

speedups for numerical factorization given in Figure 7.17 and Figure 7.18, we observe that the speedups for the SES solver package are higher than the ones for the PARDISO solver. PARDISO employs a dynamic scheduling, whereas, in the SES solver package we employ a subtree to thread mapping based on the estimated subtree factorization times. As shown in Figure 7.17, our static mapping generally gives better speedup values compared to the PARDISO's dynamic scheduling for the numerical factorization phase. For the triangular solution phase, on the other hand, the difference between the SES and PARDISO speedups is not as significant as it is for the numerical factorization phase.

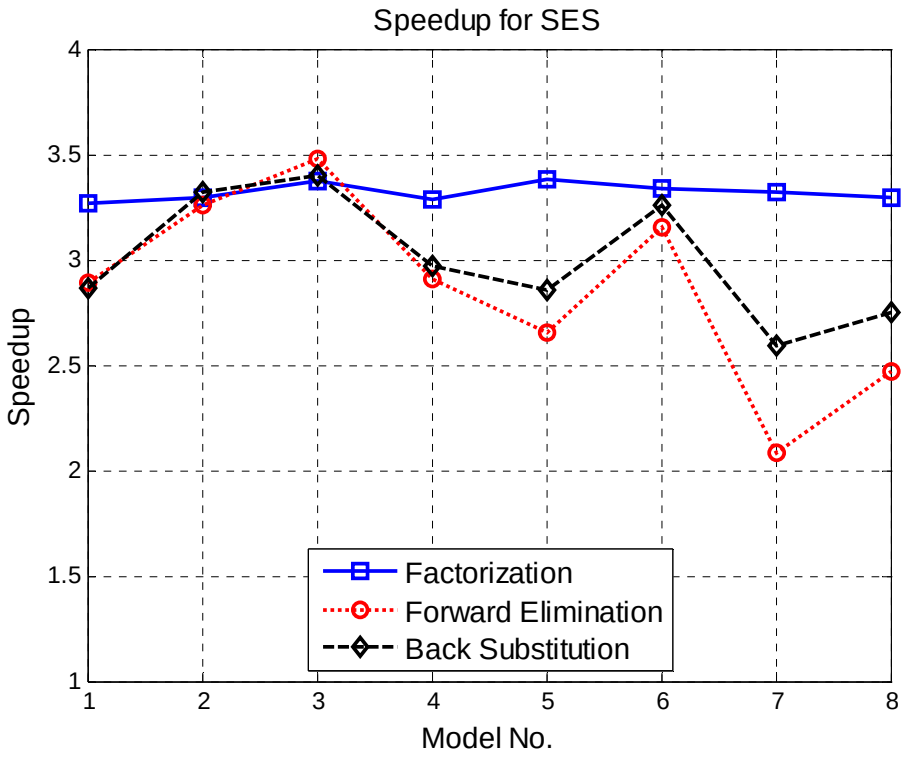


Figure 7.17: Speedup for four-thread execution of the SES solver package for 100 RHS vectors, the benchmark suite with 8 large test problems, HMETIS ordering

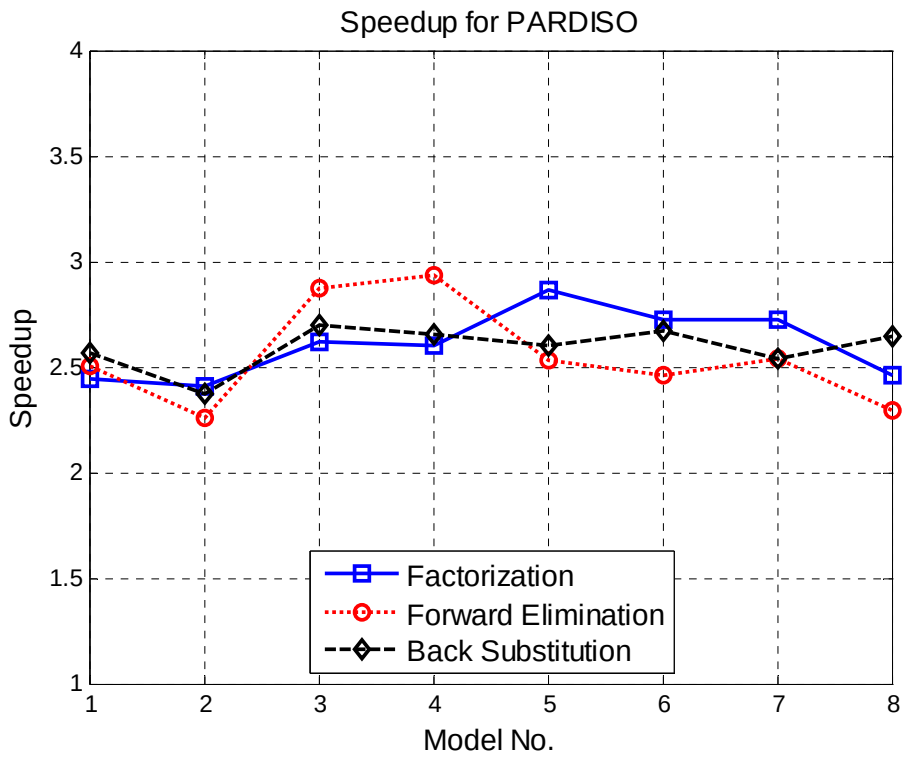


Figure 7.18: Speedup for four-thread execution of the PARDISO solver package for 100 RHS vectors, the benchmark suite with 8 large test problems, HMETIS ordering

Next, we investigate the imbalance between the workloads statically assigned to the threads in the mapping algorithm. After a mapping is found for the threads, no additional load balancing is performed. However, it is possible to minimize workload imbalances by reassigning some of the tree nodes assigned to a thread with a high workload. We can potentially use the estimated factorization times for such a load balancing. Figure 7.19 shows the idealized speedups of such a load balancing approach. Here, we assume that the subtree factorization workloads can be perfectly balanced between the threads. Figure 7.19 also shows the speedups of numerical factorization for the current implementation of the SES solver package. As shown in Figure 7.19, the multithreaded factorization performance for some test problems can be somewhat improved if we assume that a perfect load balancing can be performed. However, the performance improvements are not drastic even with an ideal workload balancing

scheme. Figure 7.19 is for HMETIS matrix ordering program and the results are similar for the AMF matrix ordering program for the benchmark suite with 8 large test problems. Considering the overhead and complexity that will be introduced by the implementation of a load balancing algorithm, it is questionable whether the implementation of additional load balancing is necessary.

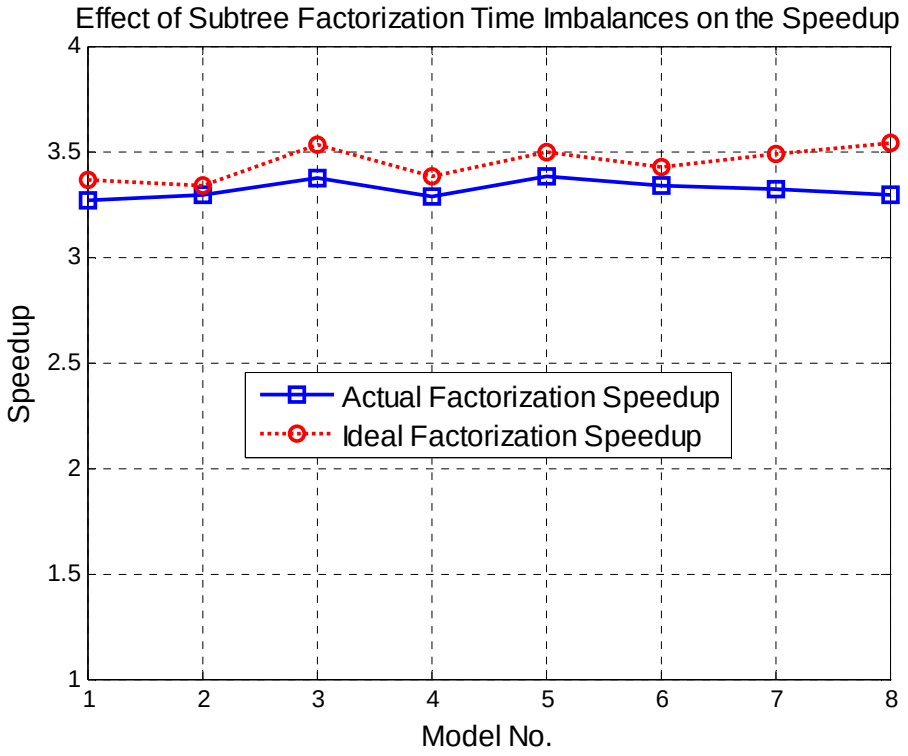


Figure 7.19: The effect of subtree factorization time imbalances on the factorization speedups for the benchmark suite with 8 large test problems using HMETIS ordering

7.2 Out-of-core Solver

The performance of out-of-core solver is demonstrated on the 8 very large test problems given in Chapter 2. The properties of the test problems are given again in Table 7.6. Although the out-of-core solver may allow numerical solution and triangular factorization of even larger 2D problems, the memory required for the preprocessing phase is the main restriction on further increasing the size of 2D test problems. For the out-of-core solver, the forward elimination is performed just after the factors are computed for a frontal matrix. As stated previously, this scheme requires fewer disk reads. The I/O operations are performed asynchronously in order to overlap disk operations with the computations. Tree-level parallelism is not exploited for the out-of-core multifrontal solver since this typically degrades the performance due to excessive I/O requests of the threads assigned to the subtrees. Therefore, parallelism is exploited only at the dense matrix level by the use of four-thread BLAS/LAPACK subroutines (MKL).

Model Name	Number of Dofs	Non-zero for HMETIS	Memory for Factors (GBytes)	Flop for HMETIS
1.Q2DVL1	4,830,000	6.12E+08	4.56	7.32E+11
2.Q2DVL2	4,500,000	4.94E+08	3.68	5.06E+11
3.F2DVL1	9,990,000	9.24E+08	6.88	7.20E+11
4.F2DVL2	9,940,000	1.01E+09	7.51	1.17E+12
5.S3DVL1	1,150,000	1.37E+09	10.19	5.66E+12
6.S3DVL2	2,210,000	1.56E+09	11.61	5.37E+12
7.F3DVL1	4,000,000	2.30E+09	17.12	7.07E+12
8.F3DVL2	1,650,000	1.35E+09	10.08	5.31E+12

Table 7.6: 8 very large test problems used for evaluating the performance of the out-of-core solver.

The performance of an out-of-core scheme can be evaluated effectively by comparing the out-of-core performance with the in-core performance. However, these very large problems cannot be solved using only main memory. Therefore, we used the

estimated serial factorization times in order to evaluate the performance of the out-of-core solver. Figure 7.20 shows the out-of-core solver solution times in terms of estimated serial factorization time. For the results shown in Figure 7.20, the solution is performed for 10 RHS vectors and the serial factorization time estimation assumes an infinite memory. As shown in Figure 7.20, the out-of-core performance for 2D test problems is worse than what it would be for a serial in-core factorization with an infinite amount of memory. According to the relative out-of-core performance given in Figure 7.20, solving the 2D problems on a machine with a larger memory will perform significantly better than the out-of-core solver. For example, Model 3 can be solved about 10× faster than the speed of out-of-core solver by employing a multithreaded in-core solver on a machine with sufficient amount of memory (assuming that in-core solver has a speedup of 3). On the other hand, the out-of-core SES solver performs better than the estimated serial factorization time for 3D test problems (Models 5 to 8). Therefore, for 3D problems, the speed increase by the use of a system with sufficient amount of memory is not as drastic as it is for the 2D problems.

The relatively low performance on the 2D test problems is due to the large numbers of I/O operations for each factorization operation. In order to obtain high performance from an out-of-core solver, a sufficient number of operations should be performed between I/O requests. Otherwise, the performance gap between the CPU and disk hinders the performance. Figure 7.21 shows the non-zero to flop ratios for the very large test problems. A high non-zero to flop ratio indicates that the number of disk accesses for each factorization operation is large, which degrades the performance of the out-of-core solver. The non-zero to flop ratio given in Figure 7.21 qualitatively follows the SES out-of-core solution performance relative to in-core solver performance given in Figure 7.20.

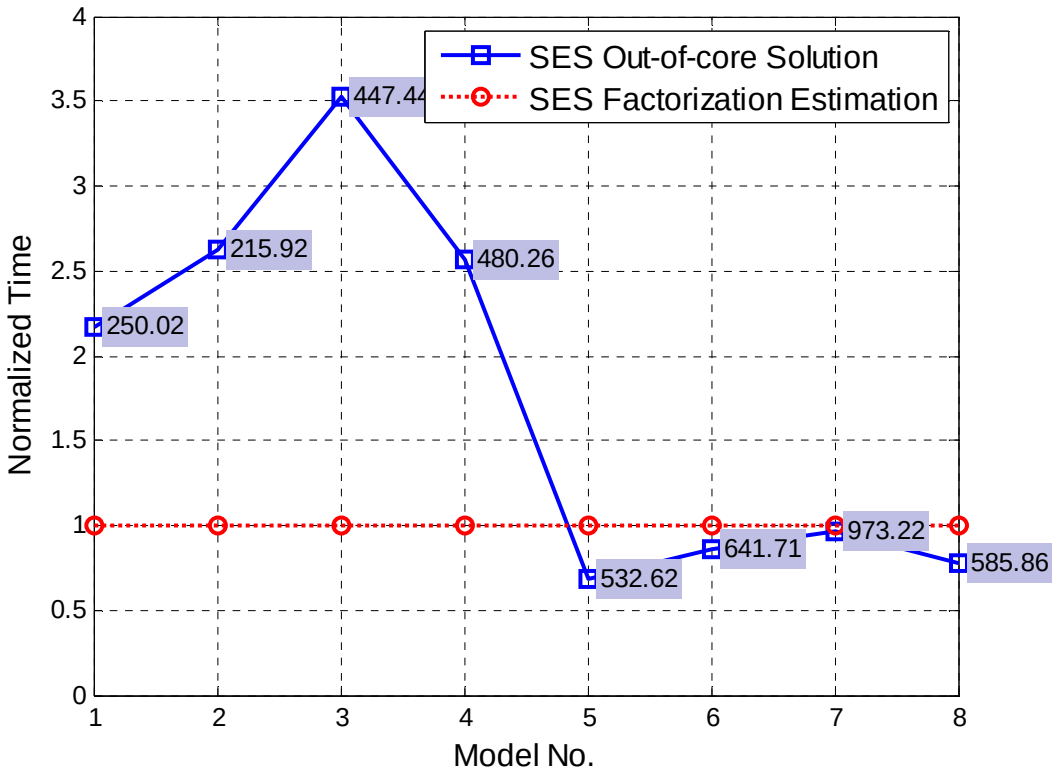


Figure 7.20: For very large test problems, SES out-of-core solution time (factorization plus triangular solution with 10 RHS vectors) given in terms of estimated single-thread factorization time assuming infinite memory

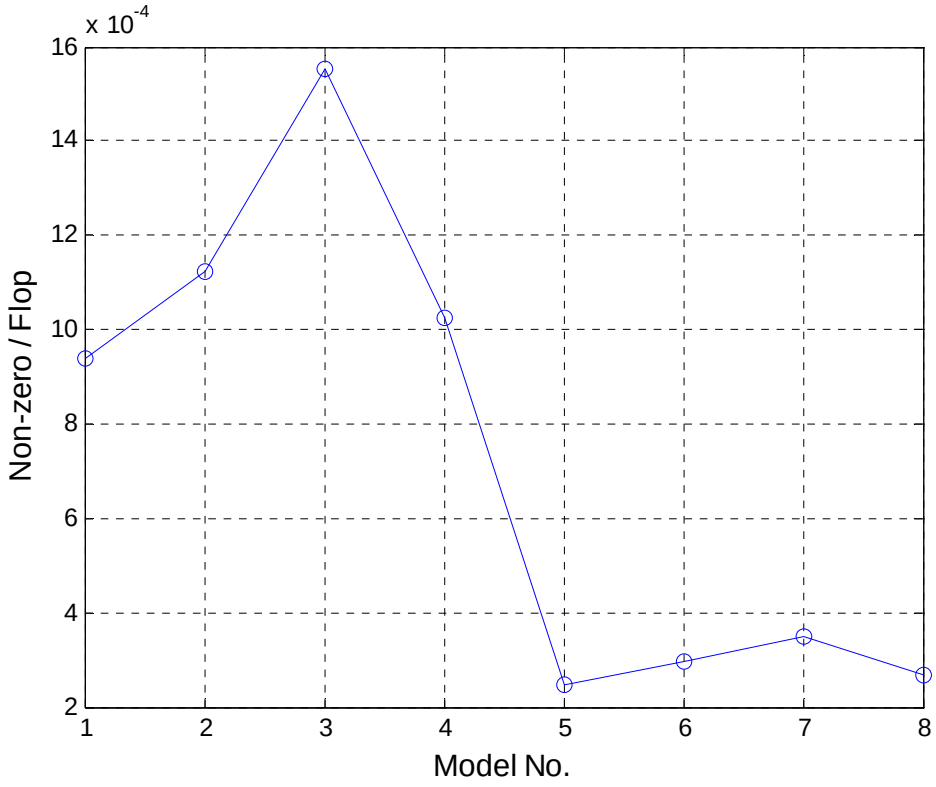


Figure 7.21: For very large test problems, the ratio of non-zero (in factorized stiffness matrix) to flop (floating point operations required for factorization).

The benchmark suite with very large test problems is also solved with the PARDISO solver package. However, PARDISO failed to solve 6 out of 8 of the very large test problems exiting with an error message saying that there is not enough memory for the preprocessing phase. PARDISO finds the solution only for Models 5 and 8, which are the models with the fewest dofs among the 8 very large test problems. For Models 5 and 8, the PARDISO solution takes 1655.32 sec and 1530.06 sec respectively while the SES solution takes 532.62 and 585.86 respectively as shown in Figure 7.20. In other words, the PARDISO solutions are about 3 times slower for the two test problems.

CHAPTER 8

FACTORIZATION USING A GPGPU

8.1 GPGPU Computing

Recently, graphical processing units have become available for general purpose computing (GPGPU). Data-parallel computations can be accelerated by using a system with GPGPUs. The GPGPU can be treated as a co-processor to perform some of the factorization tasks in the multifrontal method. The subsequent section discusses the results from a preliminary investigation of the performance gains obtained using a GPGPU for the partial factorization.

8.2 Partial Factorizations on GPGPUs

The partial factorizations of the frontal matrix are the most time consuming component of the Cholesky Decomposition with the multifrontal method. GPGPUs are ideally suited for performing these computations since they typically have a high ratio of computations per data access. The computation of the off-diagonal factors and Schur complement can be performed on the GPGPU using the corresponding CUBLAS library functions [152] to replace the computations performed on CPU using the MKL library. The diagonal factors are computed on CPU (host) since this is a LAPACK subroutine. The update matrix operations and assembly of FE matrices should also be performed on the host since these are memory bound operations including branches which are not suitable for GPGPU computing.

Preliminary numerical experiments are performed in order to evaluate potential performance gains by the implementation of a GPGPU accelerated multifrontal solver. The system used for the numerical experiments has an Intel Xeon X5550 Quad-core Nehalem processor as the host and an Nvidia S1070 Tesla unit as the GPGPU device.

Tesla unit has 960 processors and the peak double-precision performance of the unit is between 311 and 345 Gflop/sec. Each CPU core runs at a speed of 2.66 GHz with hyper-threading disabled. Intel MKL libraries [6] are used for the comparison of the GPGPU partial factorization times with the host partial factorization times. The experiments are performed using one host CPU core only.

The performance of multifrontal method accelerated with the GPGPU is compared with the CPU performance. Computing the off-diagonal factors and Schur complement on the GPGPU requires copying the already computed diagonal factors and assembled off-diagonal factors to the GPGPU. This copying of the data to the GPGPU may take a significant amount of time. Figure 8.1 shows the speed of partial factorization without including the time required to transfer the data to the GPGPU. As shown in Figure 8.1, performing partial factorization on the GPGPU outperforms single-core factorization for frontal matrices having more than 400 columns. Furthermore, GPGPU partial factorization is approximately three times faster than the single-core counterpart for frontal matrices having more than 4000 columns. Figure 8.2 shows the speed of partial factorization including the time required for the data transfer. As shown in Figure 8.2, the effective speed of GPGPU partial factorization is reduced when the data transfer time is included. The speed decrease in percentage is larger for smaller frontal matrices due to the data transfer latency and small amount of computations on GPGPU.

As shown in Section 4.4.1, the partial factorization with four threads can give a speedup larger than 3.5 over the single-threaded counterpart for sufficiently large matrices (matrices in the order of thousands). Therefore, the use of four-thread BLAS3 kernels for partial factorization is likely to outperform (or at least yield the same performance as) the GPGPU partial factorization.

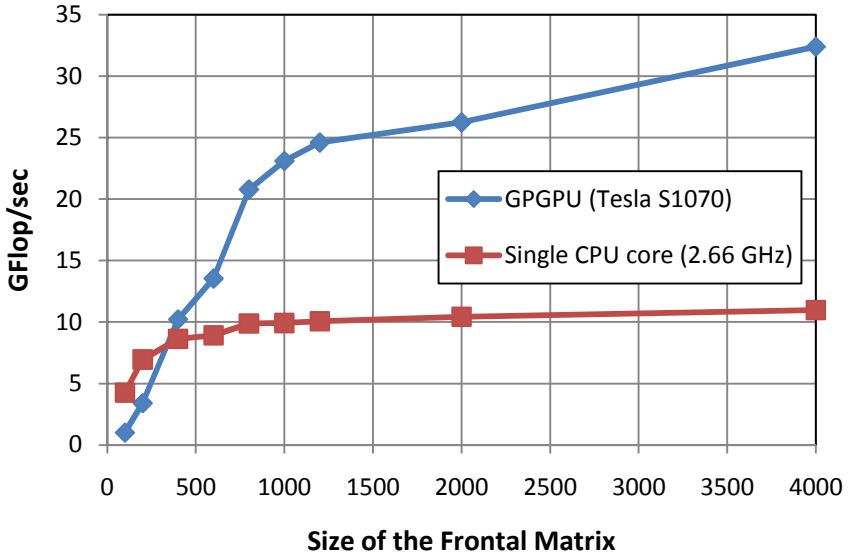


Figure 8.1: The speed of GPGPU and singe CPU core partial factorization without considering the data transfer time between host and device. For the frontal matrices, the ratio of number of remaining variables to number of eliminated variables is three.

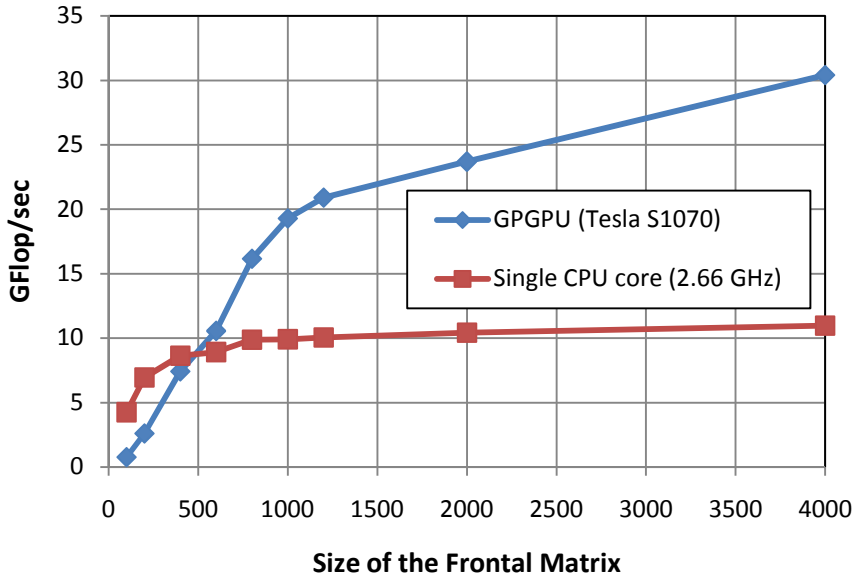


Figure 8.2: The speed of GPGPU and single CPU core partial factorization including the data transfer time between host and device. For the frontal matrices, the ratio of number of remaining variables to number of eliminated variables is three.

The performance of GPGPU accelerated partial factorization is investigated for a set of test problems shown in Table 8.1. As stated previously, only the off-diagonal factors and the Schur complement are computed on the GPGPU. The diagonal factors are computed on the host CPU. As shown in Table 8.1, computation of the Schur complement is has the largest number of floating point operations for the partial factorization. The operation counts given in Table 8.1 are for the pivot ordering found by the hybrid matrix ordering program in the METIS library [69]. The assembly trees are constructed for the test problems and the partial factorization operations associated with the assembly tree nodes are performed using the GPGPU and a single CPU core.

Figure 8.3 compares the speed for performing partial factorization operations on a single CPU core and a GPGPU. Figure 8.3 also shows the effective speed of GPGPU factorization if the time required for copying the data to GPGPU is also included. As shown in Figure 8.3, the speed of partial factorization on a GPGPU increases as the model size increases. The GPGPU speedup over a single CPU core can be as high as 3 for the largest test problem. However, the use of four-thread BLAS kernels is likely to give at least the same speed as the GPGPU partial factorization.

Model Name	Diagonal Factors (GFlop)	Off-diagonal Factors (GFlop)	Schur Complement (GFlop)	Total Partial Factorization (GFlop)
s15×15×50	4.19	3.68	10.95	18.83
f15×15×50	11.52	10.78	54.16	76.46
s30×30×30	25.93	22.93	91.86	140.72
s100×50×13	34.13	68.46	212.36	314.94
f30×30×30	64.31	83.71	396.45	544.48
s50×50×50	544.89	498.16	1934.88	2977.92
f45×45×45	993.80	897.08	5031.14	6922.03

Table 8.1: Test problems used to evaluate the performance of GPGPU accelerated partial factorization

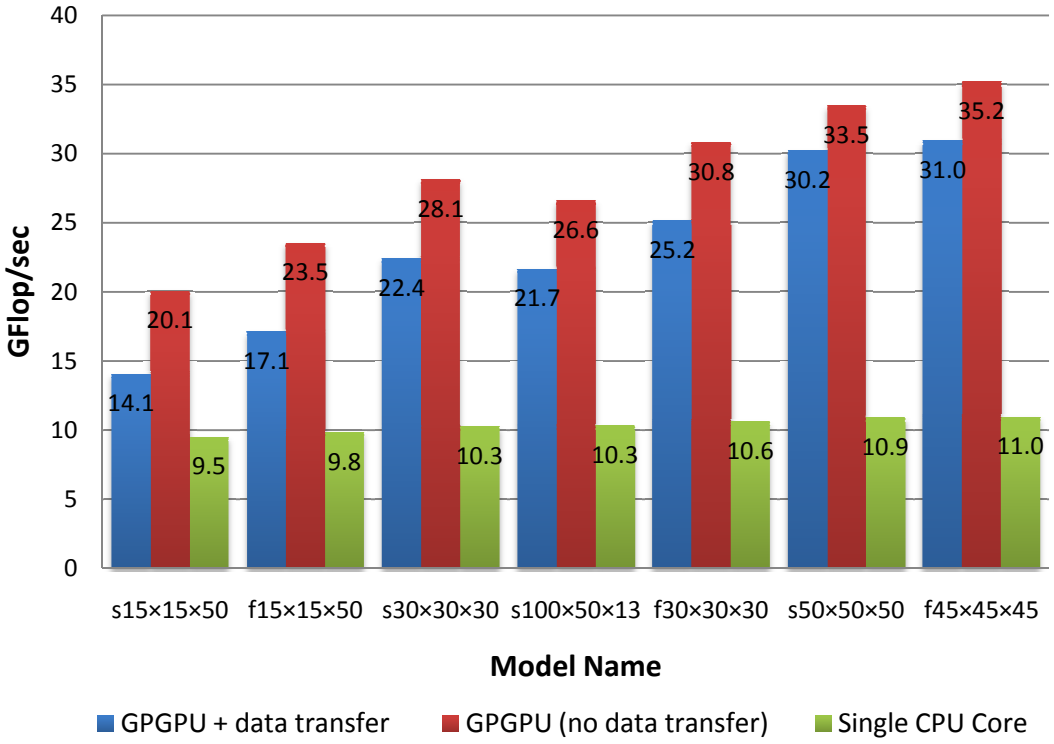


Figure 8.3: Performance of GPGPU accelerated partial factorization for the test problems given in Table 8.1.

Preliminary performance analyses illustrate that currently, the GPGPU acceleration does not offer a significant improvement over a modern multi-core CPU. Nevertheless, the results shown in this chapter is for double-precision floating point which is about 10 times slower than the single-precision floating point counterpart for the Tesla S1070 used in the experiments [153]. Double-precision floating point operations are only half the speed of the single-precision counterparts for the CPUs and are necessary for FE computations. According to NVIDIA [154], double-precision arithmetic for upcoming NVIDIA FERMI processor is about four times faster than the previous processor generation. For the next generation GPGPUs, the performance difference between double-precision and single-precision arithmetic is nearly the same as the CPU counterpart [155].

CHAPTER 9

SUMMARY AND FUTURE DIRECTIONS

9.1 Summary and Conclusions

This study proposed and developed a direct solution procedure which exploited the parallelism that exists in current symmetric multiprocessing (SMP) multi-core processors. The performance of the direct solver developed in this study demonstrated on a large suite of test problems, including problems with 100 load cases. A sparse direct solver is typically composed of four phases: preprocessing phase, analysis phase, numerical factorization phase, and triangular solution phase. Approaches to improve the performance of all four phases were discussed. Furthermore, the contribution of each phase to the overall execution time of direct solver was studied.

The first phase of a sparse direct solver, the preprocessing phase, determines a pivot-ordering which attempts to minimize the memory and CPU time requirements of the solution. There are several matrix ordering algorithms to find a pivot-ordering. The performance of alternative matrix ordering programs were evaluated using large number of 2D and 3D FE test problems. The effect of the different matrix ordering programs on both serial and parallel factorization times was determined. As noted in the previous research, the numerical experiments showed that local matrix ordering programs are sensitive to the initial node numberings. Among the initial node numberings investigated, numbering the nodes according to the node coordinates usually minimized the memory and CPU time requirements for the factorization and triangular solution. The improvements in CPU time and memory were significant for the local matrix ordering programs AMF and MMD but moderate for the local matrix ordering AMD. There was no single initial node numbering that consistently yielded favorable factorization times for the hybrid matrix ordering programs, HAMF and HMETIS. However, the quality of

the pivot-orderings could be improved by choosing the best pivot-ordering among the results from random initial node permutations.

In this study, the matrix-ordering programs AMF, MMD, CAMD, HMETIS and HAMF were used to find the pivot-ordering for the factorization and triangular solution. Among the matrix ordering programs investigated in this study, AMF usually minimized the CPU time and memory requirements for the serial solution of 2D test problems. Furthermore, AMF performed well for 3D test problems with 2D-like geometry and 3D models with small average node adjacency, models for which nodes have a small number of adjacent nodes. For the remaining 3D test problems, the hybrid matrix ordering program HMETIS yielded favorable CPU time and memory requirements.

In previous studies, hybrid matrix orderings were preferred for the parallel factorization instead of local matrix orderings since the assembly trees produced by a local ordering are not suitable for exploiting tree-level parallelism. However, this study showed that although it is true that the local orderings are relatively less suitable for parallel processing, a local matrix-ordering program can still minimize the parallel factorization and triangular solution time for current SMP multi-core processors with a small number of cores. For example, the hybrid matrix ordering HMETIS yielded better parallel factorization times for only some but not all of the test problems for which the local matrix ordering AMF gave the best serial factorization times. A preprocessing scheme that selects the best pivot-ordering among the results of hybrid and local matrix ordering programs was developed. The developed scheme performs the selection based on the estimated parallel factorization times.

A coarsening scheme was proposed to reduce the execution time of the matrix ordering programs and analysis phase. The impact of alternative mesh coarsening schemes on the quality of the pivot-orderings is determined. The numerical experiments showed that the coarsening scheme yielded better factorization times for the majority of the 2D test problems compared to using the original mesh in the matrix ordering

programs. A preprocessing scheme that employs alternative coarsening schemes to minimize the CPU time and memory requirements of the solution was developed.

The multifrontal method was adopted for parallel factorization and triangular solution of FE problems. Both tree-level and dense matrix level parallelism were exploited to improve the efficiency of the parallel solver. A mapping algorithm that automatically chooses between the two levels of parallelism was proposed, which attempts to minimize the parallel execution time based on the performance model constructed for a SMP multi-core processor. The performance model considers the contributions from the update matrix assembly times, FE assembly times, and partial factorization times for predicting the overall factorization time. The developed performance model accurately predicted the serial factorization times of the test problems. However, parallel factorization took longer than the predicted execution time. This unanticipated performance degradation in parallel factorization is mainly due to the resource contention (contention of memory bus or shared caches) on SMP multi-core processor. It is conjectured that resource contention may hinder the scalability for SMP multi-core processors with larger number of cores. To offset this degradation, the performance model and the mapping algorithm can be modified in the light of the performance degradation measured for a machine.

The performance model developed in this study can be used to choose among pivot-orderings produced by alternative matrix ordering programs or preprocessing strategies. For the test problem f75×150×5, selecting a pivot-ordering based on the factorization operation count yielded factorization times significantly worse than a selection based on the estimated factorization times. It was shown that the factorization time is affected by the number of update matrix operations and size of the frontal matrices. The performance model developed in this study incorporated these factors to predict factorization times accurately.

General purpose direct solver packages usually work with an assembled coefficient matrix. Therefore, the stiffness matrix should be assembled prior to the execution of the solver package. The assembly of the stiffness matrix may take significant time and also require storage of the assembled stiffness matrix. The developed solver package does not require an assembled stiffness matrix and it works with the element stiffness matrices. The frontal matrices are assembled in parallel on each core. For reducing the overall execution time, performing the assembly in parallel becomes especially important as the execution times of the factorization and triangular solution phases are decreased by employing parallel algorithms.

The performance of triangular solution phase may be overlooked in the development of a sparse solver since the triangular solution time is often insignificant compared to the numerical factorization. Nevertheless, the numerical experiments showed that the triangular solution time for multiple RHS vectors may be comparable to the factorization time. A triangular solution algorithm that is efficient for solution with a large number of RHS vectors was developed. The efficiency of optimized BLAS3 kernels was extended to the triangular solution phase by performing the forward elimination and back substitution operations on dense frontal matrices.

In the multifrontal method, the factors can be written to a secondary storage as soon as they are calculated. An out-of-core solver that takes advantage of this property of the multifrontal method was developed in order to reduce the memory requirements of the solver. The factors were written to the disk asynchronously to overlap factorization computations with I/O. The performance of in-core and out-of-core versions of the solver was evaluated using test problems with various sizes and element types. The performance of the developed solver was demonstrated by comparing the execution times with a commonly used shared memory solver, PARDISO. The developed code outperformed the PARDISO solver for almost every test problem. In addition, some of the large problems that could not be solved with PARDISO due to memory requirements could be solved

using the developed solver. A test problem with more than 10 million dofs was solved on a low price desktop computer using the out-of-core solver developed in this study.

9.2 Recommendations for Future Work

The sparse direct solver developed in this study implements several approaches to improve the performance of the sparse direct solution of finite element (FE) problems. However, high-performance sparse direct solution is an active area of research and the direct solver can hardly be considered as complete. The performance of the solver can be potentially improved by implementing existing and emerging algorithms. In addition, further research is required to develop a robust and high performance solver on future multi-core and many-core architectures. The recommendations for the future research are as follows:

- **Extend work to Eigen Solution of FE Problems**

Modal analysis of the structures requires computing eigenvectors and eigenvalues corresponding to the modal shapes and natural frequencies of the structures respectively. Finding eigenvectors and eigenvalues is a computationally intensive procedure, especially if a large number of eigenvectors and eigenvalues will be found. An eigen solution scheme which exploits parallelism in multi-core processors and is efficient for FE problems can be developed.

- **Many-core and heterogeneous architectures**

It is most likely that the demand for increased performance will be met by increasing the number of processing units in computers. A parallel solver which takes advantage of the emerging parallel computer architectures is crucial for the efficient solution of FE models progressing in complexity and size. The developed solver package can be modified for an efficient solution on NUMA architectures. For the NUMA architecture, the mapping algorithm should be modified to guarantee that the tasks assigned to the processors primarily access local memory. Furthermore, a sparse solver

that makes use of the GPGPUs may offer a performance improvement. Preliminary investigations were performed for performing some of the partial factorization tasks on GPGPUs. A sparse solver that treats GPGPUs as co-processors can be developed as shown in the preliminary investigations given in Chapter 8.

- **Improve the performance of the analysis and preprocessing phases**

The developed code is designed to allow for experimenting with various algorithms in the preprocessing and analysis phases. The performance of the preprocessing and analysis phases can be improved at an expense of reduced flexibility. It may even be possible to improve the performance without compromising the flexibility. The algorithms proposed by Liu [73] and Gilbert et al. [138] can be implemented to improve the performance of the analysis phase. To our knowledge, these are the best known algorithms for building the elimination tree and calculating the non-zeros in the factors. Furthermore, the matrix ordering can be performed in parallel by employing the multithreaded version of the SCOTCH library. It is crucial to parallelize the preprocessing and analysis phases of a direct solver since these two phases become a bottleneck as the numerical factorization and triangular solution times are decreased.

- **A strategy that automatically selects the matrix ordering program**

The numerical experiments with matrix ordering programs show that there is no single program that gives the best results for all FE problems. A strategy that selects the matrix ordering program that is most suitable for an input structure will reduce the overall execution time of the sparse direct solver. The model features such as model dimensionality and average node adjacency can be used in order to predetermine the most favorable matrix ordering program for an input FE problem. Further numerical experiments are required to determine other model features that will help to construct such an automatic selection strategy. Moreover, different matrix ordering programs may be desirable for different regions of the FE model. For example, 2D-Like regions in a 3D FE model, such as diaphragms, can be efficiently pre-processed with the local ordering

algorithm AMF. A preprocessing strategy that chooses the best matrix ordering program for different regions of the FE model also has the potential to further reduce the factorization times.

- **Improvements to the mapping algorithm**

As discussed previously, the subtree factorization times are larger than the anticipated times. A mapping algorithm that considers the performance degradation of the subtree factorization will potentially yield better parallel factorization times. Furthermore, the degree of tree-level parallelism may be reduced to prevent associated resource contentions. Instead of tree-level parallelism, dense matrix level parallelism can be exploited for the threads assigned to the subtrees in order to reduce memory bus contention. For example, for a quad-core processor, only two threads could be spawned to factorize independent subtrees instead of four threads and two-thread BLAS/LAPACK kernels would be called for the partial factorization tasks on independent subtrees.

- **A program for automatic construction of the performance model**

The mapping algorithm relies on the performance model constructed for the SMP multi-core processors. Presently, this is done manually by performing test runs on the test system. For a complete solver implementation, a program that automatically constructs the performance model for any computer is required. The program will perform a limited test runs with BLAS/LAPACK kernels to determine the partial factorization performance. Furthermore, the speed of update matrix operations and assembly of FE matrices can be determined by executing benchmark codes which represent these operations.

- **Improvements to the numerical factorization and solution phases**

Currently, the multifrontal scheme developed in this study does not employ a memory minimizing scheme for the update matrix stack. The memory minimizing schemes proposed by Guermouche and L'excellent [55] can be implemented to reduce the active memory requirements of the multifrontal solver. Furthermore, instead of allocating

separate memory locations for the frontal matrix and update matrix stack, the memory can be shared between the two in order to reduce the memory footprint of the solver. The triangular solution phase will also benefit from these modifications. Currently, the triangular solution phase is optimized for multiple RHS vectors. A triangular solution that is tuned for solution of single RHS vector can improve the triangular solution performance for non-linear or transient analyses.

- **Improvements to the out-of-core solver**

Currently, the out-of-core solver writes only the computed factors to disk. The out-of-core solver could also write the update matrix stack and frontal matrix to the disk. The update matrix stack and frontal matrix can be written to the disk asynchronously since asynchronous I/O is proved to be efficient for the current out-of-core implementation. Furthermore, the solver can automatically determine when to switch to the out-of-core mode based on the available memory and memory required for the solution of an input problem.

- **Test problems with mixed elements**

All test problems used to evaluate the performance of various algorithms are composed of a single FE type. The test problems with mixed elements, such as frames and plates, can be added to the test suites.

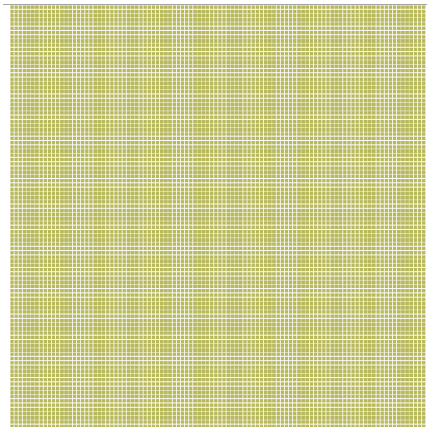
APPENDIX A:

TEST PROBLEMS

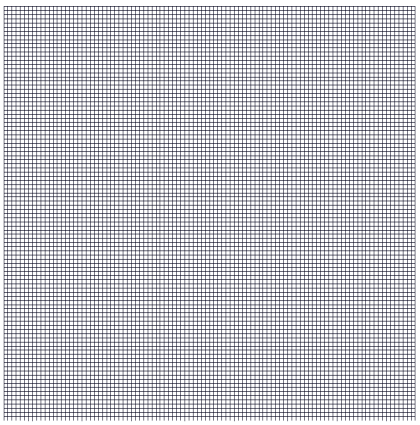
A.1 Test Problems with Regular Geometry

The geometry of regular test problems is rectangular in 2D and prismatic in 3D. In other words, the number of elements in x, y and z directions is the same at any location. Figure A.1 shows the geometry of the example models with regular geometries. Figure A.2 and A.3 shows the non-zero pattern for the 2D test problems q100×100 and f100×100 respectively. Figure A.4 and A.5 shows the non-zero pattern for the 3D test problems s10×10×10 and f10×10×10 respectively.

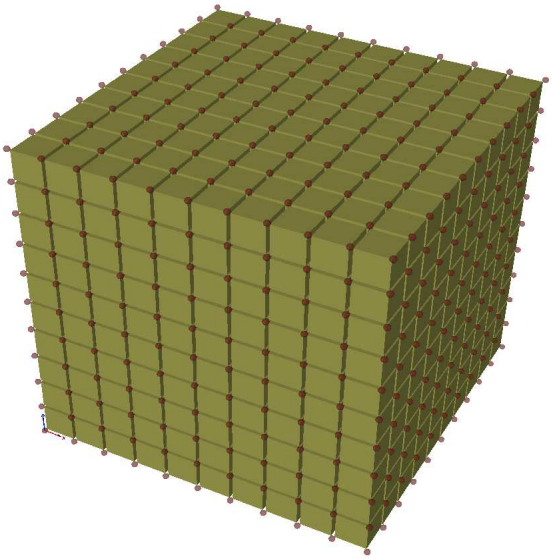
Table A.1 gives the properties of the 2D test problems with quadrilateral elements. Table A.2 gives the properties of the 2D test problems with frame elements. Table A.3 gives the properties of the 3D test problems with 8 node solid elements. And finally, Table A.4 gives the properties of the 3D test problems with 3D frame elements. The last column in these tables shows the number of non-zero entries in the lower diagonal stiffness matrix. The non-zero is computed by assuming that all entries of element stiffness matrices are non-zero.



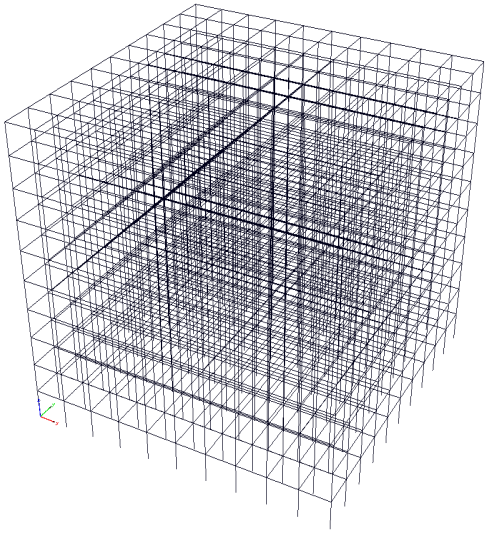
q100x100



f100x100



s10x10x10



f10x10x10

Figure A.1: Selected test problems with regular geometry.

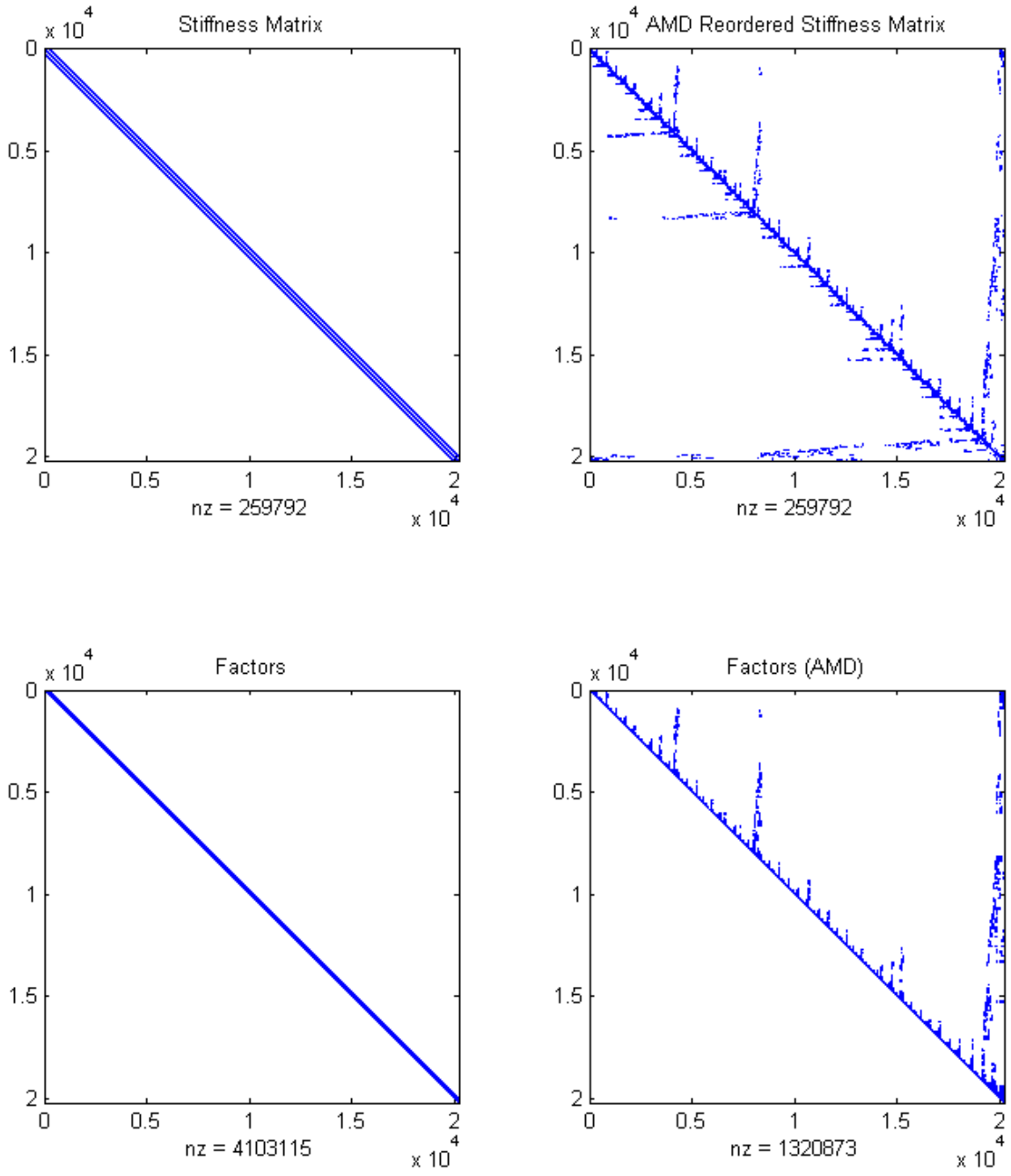


Figure A.2: Non-zero patterns for the test problem q100×100 (original ordering and AMD matrix ordering). The non-zero patterns for the upper diagonal factors are also given at the bottom.

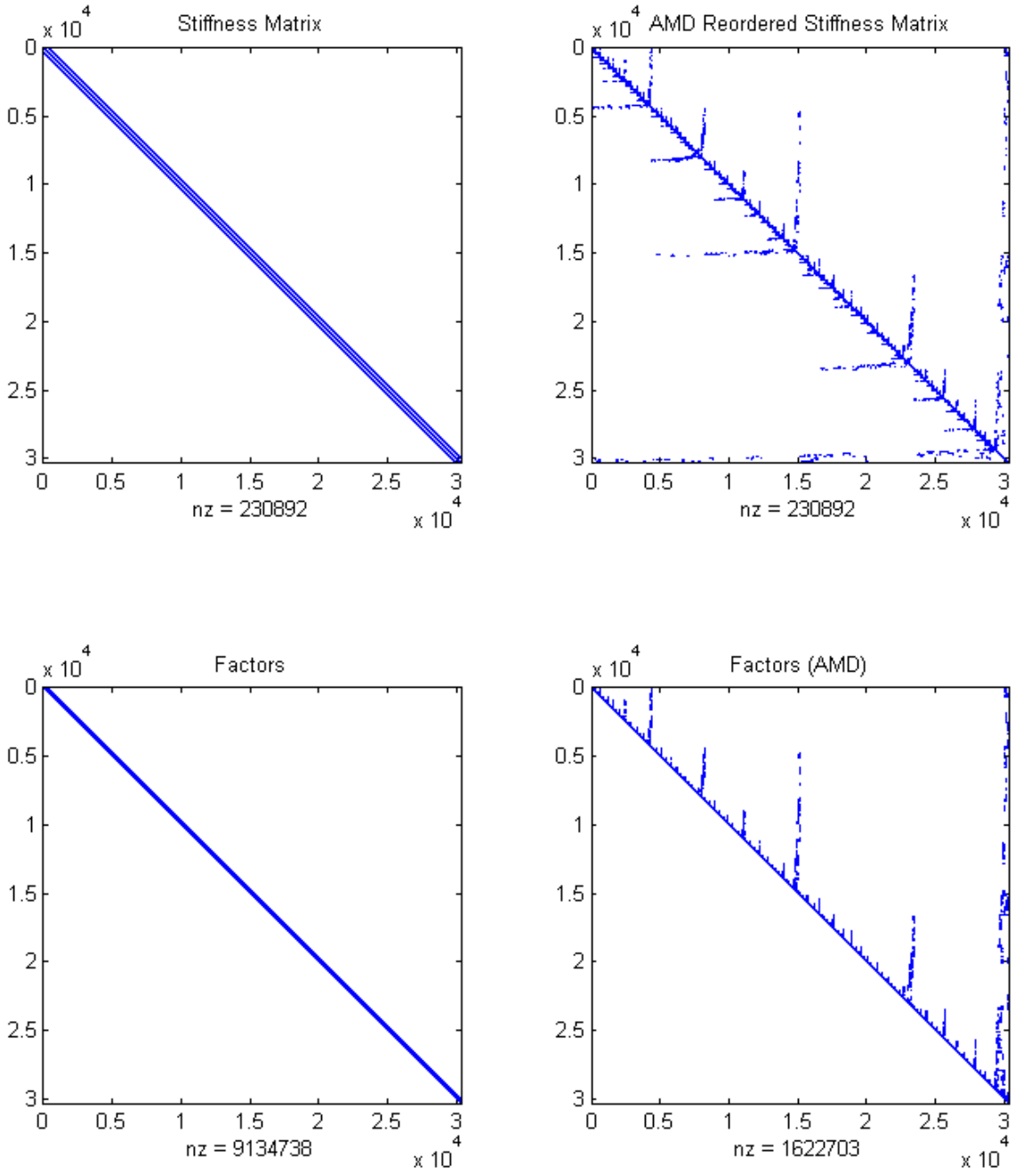


Figure A.3: Non-zero patterns for the test problem f100×100 (original ordering and AMD matrix ordering). The non-zero patterns for the upper diagonal factors are also given at the bottom.

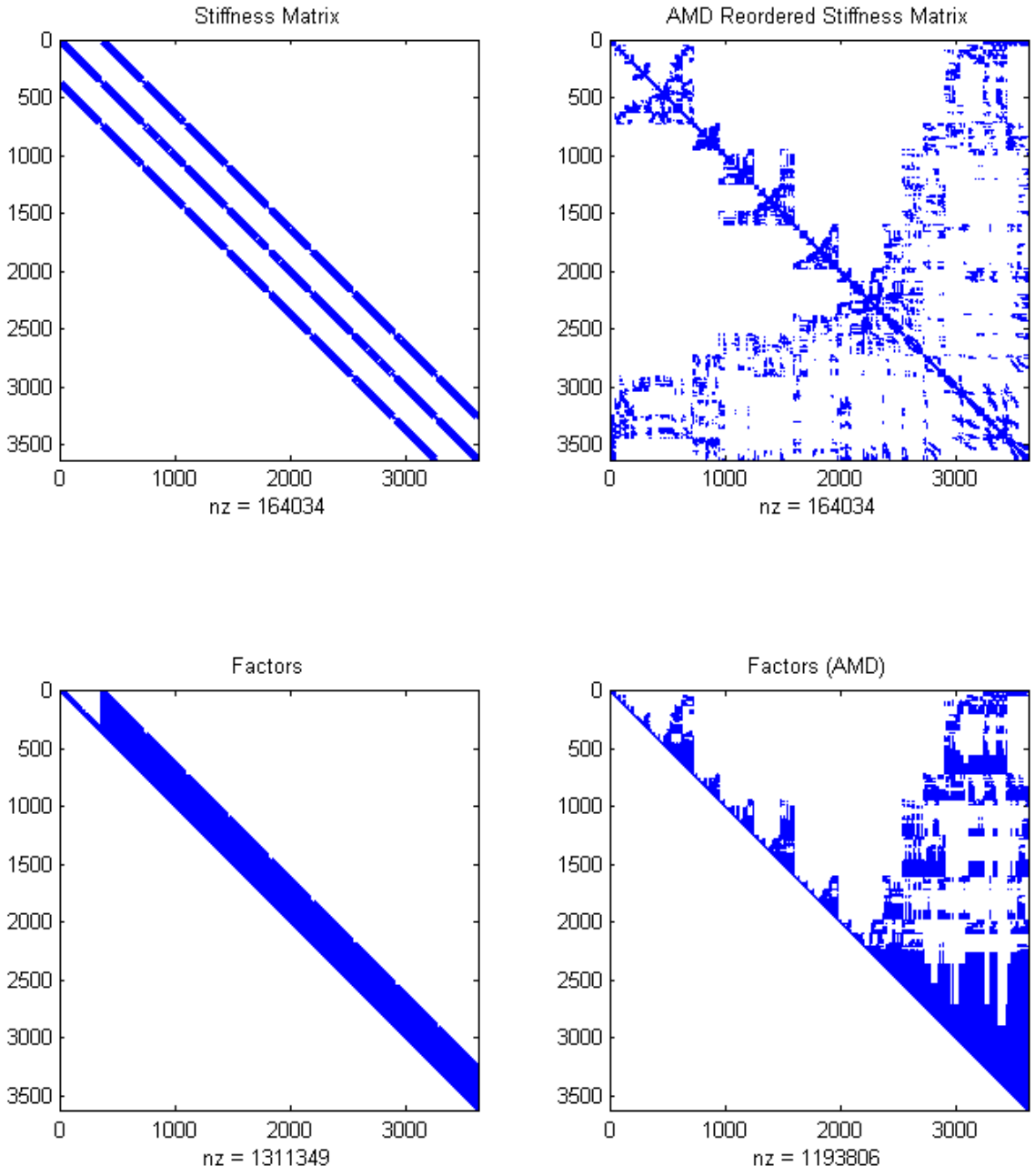


Figure A.4: Non-zero patterns for the test problem s10×10×10 (original ordering and AMD matrix ordering). The non-zero patterns for the upper diagonal factors are also given at the bottom.

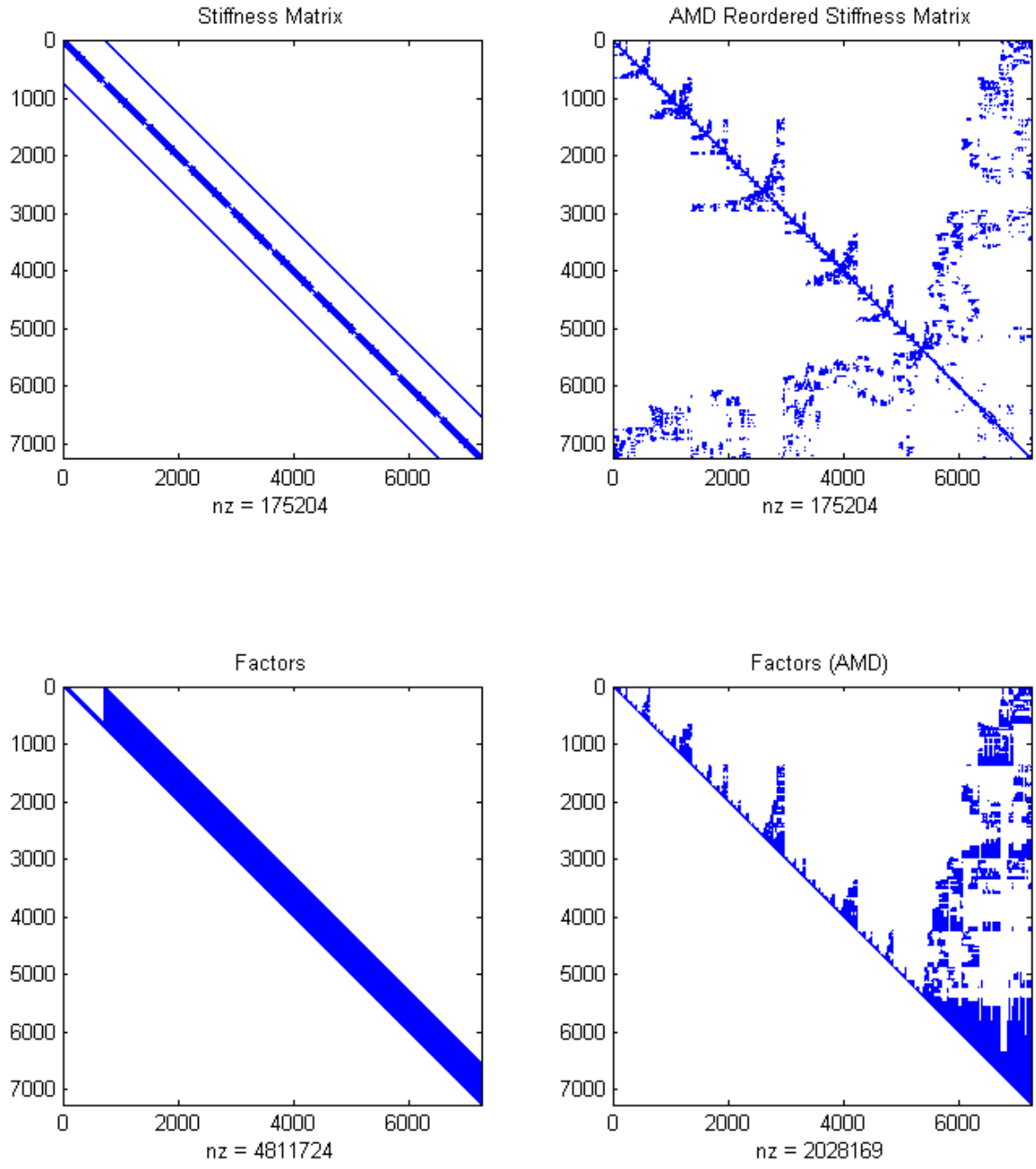


Figure A.5: Non-zero patterns for the test problem f10×10×10 (original ordering and AMD matrix ordering). The non-zero patterns for the upper diagonal factors are also given at the bottom.

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
1	q20×1000	20000	42000	0.39
2	q20×2000	40000	84000	0.77
3	q20×3000	60000	126000	1.16
4	q20×4000	80000	168000	1.55
5	q20×5000	100000	210000	1.93
6	q20×7500	150000	315000	2.9
7	q20×10000	200000	420000	3.87
8	q30×1000	30000	62000	0.58
9	q30×2000	60000	124000	1.15
10	q30×3000	90000	186000	1.73
11	q30×4000	120000	248000	2.31
12	q30×5000	150000	310000	2.88
13	q30×7500	225000	465000	4.33
14	q30×10000	300000	620000	5.77
15	q40×1000	40000	82000	0.77
16	q40×2000	80000	164000	1.53
17	q40×3000	120000	246000	2.3
18	q40×4000	160000	328000	3.07
19	q40×5000	200000	410000	3.83
20	q40×10000	300000	615000	5.75
21	q40×15000	400000	820000	7.67
22	q50×1000	50000	102000	0.96
23	q50×2000	100000	204000	1.91
24	q50×3000	150000	306000	2.87
25	q50×4000	200000	408000	3.83
26	q50×5000	250000	510000	4.78
27	q50×7500	375000	765000	7.18
28	q50×10000	500000	1020000	9.57
29	q60×1000	60000	122000	1.15
30	q60×2000	120000	244000	2.29
31	q60×3000	180000	366000	3.44
32	q60×4000	240000	488000	4.59
33	q60×5000	300000	610000	5.73
34	q60×7500	450000	915000	8.6
35	q60×10000	600000	1220000	11.47
36	q70×1000	70000	142000	1.34
37	q70×2000	140000	284000	2.67
38	q70×3000	210000	426000	4.01
39	q70×4000	280000	568000	5.35
40	q70×5000	350000	710000	6.68
41	q70×7500	525000	1065000	10.03
42	q70×10000	700000	1420000	13.37

Table A.1: 2D quadrilateral element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
43	q80×1000	80000	162000	1.53
44	q80×2000	160000	324000	3.05
45	q80×3000	240000	486000	4.58
46	q80×4000	320000	648000	6.11
47	q80×5000	400000	810000	7.63
48	q80×7500	600000	1215000	11.45
49	q80×10000	800000	1620000	15.27
50	q90×1000	90000	182000	1.72
51	q90×2000	180000	364000	3.43
52	q90×3000	270000	546000	5.15
53	q90×4000	360000	728000	6.87
54	q90×5000	450000	910000	8.58
55	q90×7500	675000	1365000	12.88
56	q90×10000	900000	1820000	17.17
57	q100×100	10000	20200	0.19
58	q100×200	20000	40400	0.38
59	q100×300	30000	60600	0.57
60	q100×400	40000	80800	0.76
61	q100×500	50000	101000	0.95
62	q100×1000	100000	202000	1.91
63	q100×1500	150000	303000	2.86
64	q200×200	40000	80400	0.76
65	q200×300	60000	120600	1.14
66	q200×400	80000	160800	1.52
67	q200×500	100000	201000	1.9
68	q200×1000	200000	402000	3.8
69	q200×1500	300000	603000	5.71
70	q300×300	90000	180600	1.71
71	q300×400	120000	240800	2.28
72	q300×500	150000	301000	2.85
73	q300×1000	300000	602000	5.7
74	q300×1500	450000	903000	8.56
75	q400×400	160000	320800	3.04
76	q400×500	200000	401000	3.8
77	q400×1000	400000	802000	7.6
78	q400×1500	600000	1203000	11.41
79	q500×500	250000	501000	4.75
80	q500×1000	500000	1002000	9.5
81	q500×1500	750000	1503000	14.25
82	q1000×1000	1000000	2002000	18.99
83	q1000×1500	1500000	3003000	28.5

Table A.1 (cont.): 2D quadrilateral element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
84	f20×1000	41000	63000	0.49
85	f20×2000	82000	126000	0.99
86	f20×3000	123000	189000	1.48
87	f20×4000	164000	252000	1.98
88	f20×5000	205000	315000	2.47
89	f20×7500	307500	472500	3.71
90	f20×10000	410000	630000	4.95
91	f30×1000	61000	93000	0.73
92	f30×2000	122000	186000	1.47
93	f30×3000	183000	279000	2.2
94	f30×4000	244000	372000	2.94
95	f30×5000	305000	465000	3.67
96	f30×7500	457500	697500	5.51
97	f30×10000	610000	930000	7.35
98	f40×1000	81000	123000	0.97
99	f40×2000	162000	246000	1.95
100	f40×3000	243000	369000	2.92
101	f40×4000	324000	492000	3.9
102	f40×5000	405000	615000	4.87
103	f40×7500	607500	922500	7.31
104	f40×10000	810000	1230000	9.75
105	f50×1000	101000	153000	1.21
106	f50×2000	202000	306000	2.43
107	f50×3000	303000	459000	3.64
108	f50×4000	404000	612000	4.86
109	f50×5000	505000	765000	6.07
110	f50×7500	757500	1147500	9.11
111	f50×10000	1010000	1530000	12.15
112	f60×1000	121000	183000	1.45
113	f60×2000	242000	366000	2.91
114	f60×3000	363000	549000	4.36
115	f60×4000	484000	732000	5.82
116	f60×5000	605000	915000	7.27
117	f60×7500	907500	1372500	10.91
118	f60×10000	1210000	1830000	14.55
119	f70×1000	141000	213000	1.69
120	f70×2000	282000	426000	3.39
121	f70×3000	423000	639000	5.08
122	f70×4000	564000	852000	6.78
123	f70×5000	705000	1065000	8.47
124	f70×7500	1057500	1597500	12.71
125	f70×10000	1410000	2130000	16.95

Table A.2: 2D frame element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
126	f80×1000	161000	243000	1.93
127	f80×2000	322000	486000	3.87
128	f80×3000	483000	729000	5.8
129	f80×4000	644000	972000	7.74
130	f80×5000	805000	1215000	9.67
131	f80×7500	1207500	1822500	14.51
132	f80×10000	1610000	2430000	19.35
133	f90×1000	181000	273000	2.17
134	f90×2000	362000	546000	4.35
135	f90×3000	543000	819000	6.52
136	f90×4000	724000	1092000	8.7
137	f90×5000	905000	1365000	10.87
138	f90×7500	1357500	2047500	16.31
139	f90×10000	1810000	2730000	21.75
140	f100×100	20100	30300	0.24
141	f100×200	40200	60600	0.48
142	f100×300	60300	90900	0.72
143	f100×400	80400	121200	0.97
144	f100×500	100500	151500	1.21
145	f100×1000	201000	303000	2.41
146	f100×1500	301500	454500	3.62
147	f200×200	80200	120600	0.96
148	f200×300	120300	180900	1.44
149	f200×400	160400	241200	1.92
150	f200×500	200500	301500	2.41
151	f200×1000	401000	603000	4.81
152	f200×1500	601500	904500	7.22
153	f300×300	180300	270900	2.16
154	f300×400	240400	361200	2.88
155	f300×500	300500	451500	3.6
156	f300×1000	601000	903000	7.21
157	f300×1500	901500	1354500	10.82
158	f400×400	320400	481200	3.84
159	f400×500	400500	601500	4.8
160	f400×1000	801000	1203000	9.61
161	f400×1500	1201500	1804500	14.42
162	f500×500	500500	751500	6
163	f500×1000	1001000	1503000	12.01
164	f500×1500	1501500	2254500	18.02
165	f1000×1000	2001000	3003000	24.01
166	f1000×1500	3001500	4504500	36.01

Table A.2 (cont.): 2D frame element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
167	s5×5×75	1875	8100	0.26
168	s5×5×100	2500	10800	0.35
169	s5×5×125	3125	13500	0.44
170	s5×5×150	3750	16200	0.52
171	s5×5×175	4375	18900	0.61
172	s5×5×200	5000	21600	0.7
173	s5×5×250	6250	27000	0.87
174	s5×10×75	3750	14850	0.5
175	s5×10×100	5000	19800	0.67
176	s5×10×125	6250	24750	0.84
177	s5×10×150	7500	29700	1.01
178	s5×10×175	8750	34650	1.18
179	s5×10×200	10000	39600	1.35
180	s5×10×250	12500	49500	1.69
181	s5×15×75	5625	21600	0.75
182	s5×15×100	7500	28800	1
183	s5×15×125	9375	36000	1.25
184	s5×15×150	11250	43200	1.51
185	s5×15×175	13125	50400	1.76
186	s5×15×200	15000	57600	2.01
187	s5×15×250	18750	72000	2.51
188	s5×20×75	7500	28350	0.99
189	s5×20×100	10000	37800	1.33
190	s5×20×125	12500	47250	1.66
191	s5×20×150	15000	56700	2
192	s5×20×175	17500	66150	2.33
193	s5×20×200	20000	75600	2.66
194	s5×20×250	25000	94500	3.33
195	s10×10×75	7500	27225	0.98
196	s10×10×100	10000	36300	1.31
197	s10×10×125	12500	45375	1.64
198	s10×10×150	15000	54450	1.96
199	s10×10×175	17500	63525	2.29
200	s10×10×200	20000	72600	2.62
201	s10×10×250	25000	90750	3.28
202	s10×15×75	11250	39600	1.45
203	s10×15×100	15000	52800	1.94
204	s10×15×125	18750	66000	2.43
205	s10×15×150	22500	79200	2.91
206	s10×15×175	26250	92400	3.4
207	s10×15×200	30000	105600	3.89
208	s10×15×250	37500	132000	4.87

Table A.3: 3D solid element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
209	s10×20×75	15000	51975	1.92
210	s10×20×100	20000	69300	2.57
211	s10×20×125	25000	86625	3.22
212	s10×20×150	30000	103950	3.86
213	s10×20×175	35000	121275	4.51
214	s10×20×200	40000	138600	5.16
215	s10×20×250	50000	173250	6.45
216	s15×15×75	16875	57600	2.15
217	s15×15×100	22500	76800	2.88
218	s15×15×125	28125	96000	3.6
219	s15×15×150	33750	115200	4.32
220	s15×15×175	39375	134400	5.05
221	s15×15×200	45000	153600	5.77
222	s15×15×250	56250	192000	7.22
223	s15×20×75	22500	75600	2.85
224	s15×20×100	30000	100800	3.81
225	s15×20×125	37500	126000	4.77
226	s15×20×150	45000	151200	5.73
227	s15×20×175	52500	176400	6.69
228	s15×20×200	60000	201600	7.65
229	s15×20×250	75000	252000	9.57
230	s20×20×75	30000	99225	3.78
231	s20×20×100	40000	132300	5.06
232	s20×20×125	50000	165375	6.33
233	s20×20×150	60000	198450	7.6
234	s20×20×175	70000	231525	8.87
235	s20×20×200	80000	264600	10.15
236	s20×20×250	100000	330750	12.69
237	s10×10×10	1000	3630	0.12
238	s10×10×15	1500	5445	0.19
239	s10×10×20	2000	7260	0.25
240	s10×10×25	2500	9075	0.32
241	s10×10×30	3000	10890	0.39
242	s10×10×35	3500	12705	0.45
243	s10×10×40	4000	14520	0.52
244	s10×10×45	4500	16335	0.58
245	s10×10×50	5000	18150	0.65

Table A.3 (cont.): 3D solid element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
246	s10×15×15	2250	7920	0.28
247	s10×15×20	3000	10560	0.38
248	s10×15×25	3750	13200	0.47
249	s10×15×30	4500	15840	0.57
250	s10×15×35	5250	18480	0.67
251	s10×15×40	6000	21120	0.77
252	s10×15×45	6750	23760	0.87
253	s10×15×50	7500	26400	0.96
254	s10×20×20	4000	13860	0.5
255	s10×20×25	5000	17325	0.63
256	s10×20×30	6000	20790	0.76
257	s10×20×35	7000	24255	0.89
258	s10×20×40	8000	27720	1.02
259	s10×20×45	9000	31185	1.15
260	s10×20×50	10000	34650	1.28
261	s10×25×25	6250	21450	0.78
262	s10×25×30	7500	25740	0.95
263	s10×25×35	8750	30030	1.11
264	s10×25×40	10000	34320	1.27
265	s10×25×45	11250	38610	1.43
266	s10×25×50	12500	42900	1.59
267	s10×30×30	9000	30690	1.13
268	s10×30×35	10500	35805	1.33
269	s10×30×40	12000	40920	1.52
270	s10×30×45	13500	46035	1.71
271	s10×30×50	15000	51150	1.9
272	s10×35×35	12250	41580	1.54
273	s10×35×40	14000	47520	1.77
274	s10×35×45	15750	53460	1.99
275	s10×35×50	17500	59400	2.22
276	s15×15×15	3375	11520	0.41
277	s15×15×20	4500	15360	0.56
278	s15×15×25	5625	19200	0.7
279	s15×15×30	6750	23040	0.85
280	s15×15×35	7875	26880	0.99
281	s15×15×40	9000	30720	1.14
282	s15×15×45	10125	34560	1.28
283	s15×15×50	11250	38400	1.43

Table A.3 (cont.): 3D solid element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
284	s15×20×20	6000	20160	0.74
285	s15×20×25	7500	25200	0.93
286	s15×20×30	9000	30240	1.13
287	s15×20×35	10500	35280	1.32
288	s15×20×40	12000	40320	1.51
289	s15×20×45	13500	45360	1.7
290	s15×20×50	15000	50400	1.89
291	s15×25×25	9375	31200	1.16
292	s15×25×30	11250	37440	1.4
293	s15×25×35	13125	43680	1.64
294	s15×25×40	15000	49920	1.88
295	s15×25×45	16875	56160	2.12
296	s15×25×50	18750	62400	2.36
297	s15×30×30	13500	44640	1.68
298	s15×30×35	15750	52080	1.97
299	s15×30×40	18000	59520	2.25
300	s15×30×45	20250	66960	2.54
301	s15×30×50	22500	74400	2.82
302	s15×35×35	18375	60480	2.29
303	s15×35×40	21000	69120	2.62
304	s15×35×45	23625	77760	2.96
305	s15×35×50	26250	86400	3.29
306	s20×20×20	8000	26460	0.98
307	s20×20×25	10000	33075	1.24
308	s20×20×30	12000	39690	1.49
309	s20×20×35	14000	46305	1.75
310	s20×20×40	16000	52920	2
311	s20×20×45	18000	59535	2.26
312	s20×20×50	20000	66150	2.51
313	s20×25×25	12500	40950	1.54
314	s20×25×30	15000	49140	1.86
315	s20×25×35	17500	57330	2.18
316	s20×25×40	20000	65520	2.49
317	s20×25×45	22500	73710	2.81
318	s20×25×50	25000	81900	3.13
319	s20×30×30	18000	58590	2.23
320	s20×30×35	21000	68355	2.61
321	s20×30×40	24000	78120	2.99
322	s20×30×45	27000	87885	3.37
323	s20×30×50	30000	97650	3.75
324	s20×35×35	24500	79380	3.04
325	s20×35×40	28000	90720	3.48
326	s20×35×45	31500	102060	3.92
327	s20×35×50	35000	113400	4.36

Table A.3 (cont.): 3D solid element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
328	s25×25×25	15625	50700	1.92
329	s25×25×30	18750	60840	2.32
330	s25×25×35	21875	70980	2.71
331	s25×25×40	25000	81120	3.11
332	s25×25×45	28125	91260	3.5
333	s25×25×50	31250	101400	3.9
334	s25×30×30	22500	72540	2.77
335	s25×30×35	26250	84630	3.25
336	s25×30×40	30000	96720	3.72
337	s25×30×45	33750	108810	4.19
338	s25×30×50	37500	120900	4.67
339	s25×35×35	30625	98280	3.78
340	s25×35×40	35000	112320	4.33
341	s25×35×45	39375	126360	4.88
342	s25×35×50	43750	140400	5.44
343	s30×30×30	27000	86490	3.32
344	s30×30×35	31500	100905	3.89
345	s30×30×40	36000	115320	4.45
346	s30×30×45	40500	129735	5.02
347	s30×30×50	45000	144150	5.59
348	s30×35×35	36750	117180	4.53
349	s30×35×40	42000	133920	5.19
350	s30×35×45	47250	150660	5.85
351	s30×35×50	52500	167400	6.51
352	s35×35×35	42875	136080	5.28
353	s35×35×40	49000	155520	6.04
354	s35×35×45	55125	174960	6.81
355	s35×35×50	61250	194400	7.58
356	s25×25×4	2500	8112	0.26
357	s25×25×5	3125	10140	0.34
358	s25×25×6	3750	12168	0.42
359	s25×50×4	5000	15912	0.52
360	s25×50×5	6250	19890	0.68
361	s25×50×6	7500	23868	0.84
362	s25×75×4	7500	23712	0.78
363	s25×75×5	9375	29640	1.02
364	s25×75×6	11250	35568	1.25
365	s25×100×4	10000	31512	1.04
366	s25×100×5	12500	39390	1.36
367	s25×100×6	15000	47268	1.67
368	s25×125×4	12500	39312	1.31
369	s25×125×5	15625	49140	1.7
370	s25×125×6	18750	58968	2.09
371	s25×150×4	15000	47112	1.57
372	s25×150×5	18750	58890	2.03
373	s25×150×6	22500	70668	2.5

Table A.3 (cont.): 3D solid element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
374	s50×50×4	10000	31212	1.04
375	s50×50×5	12500	39015	1.35
376	s50×50×6	15000	46818	1.66
377	s50×75×4	15000	46512	1.56
378	s50×75×5	18750	58140	2.03
379	s50×75×6	22500	69768	2.49
380	s50×100×4	20000	61812	2.08
381	s50×100×5	25000	77265	2.7
382	s50×100×6	30000	92718	3.32
383	s50×125×4	25000	77112	2.59
384	s50×125×5	31250	96390	3.37
385	s50×125×6	37500	115668	4.15
386	s50×150×4	30000	92412	3.11
387	s50×150×5	37500	115515	4.04
388	s50×150×6	45000	138618	4.97
389	s75×75×4	22500	69312	2.33
390	s75×75×5	28125	86640	3.03
391	s75×75×6	33750	103968	3.73
392	s75×100×4	30000	92112	3.11
393	s75×100×5	37500	115140	4.04
394	s75×100×6	45000	138168	4.97
395	s75×125×4	37500	114912	3.88
396	s75×125×5	46875	143640	5.04
397	s75×125×6	56250	172368	6.2
398	s75×150×4	45000	137712	4.66
399	s75×150×5	56250	172140	6.05
400	s75×150×6	67500	206568	7.44
401	s100×100×4	40000	122412	4.14
402	s100×100×5	50000	153015	5.38
403	s100×100×6	60000	183618	6.61
404	s100×125×4	50000	152712	5.17
405	s100×125×5	62500	190890	6.72
406	s100×125×6	75000	229068	8.26
407	s100×150×4	60000	183012	6.2
408	s100×150×5	75000	228765	8.06
409	s100×150×6	90000	274518	9.91
410	s125×125×4	62500	190512	6.46
411	s125×125×5	78125	238140	8.39
412	s125×125×6	93750	285768	10.32
413	s125×150×4	75000	228312	7.74
414	s125×150×5	93750	285390	10.06
415	s125×150×6	112500	342468	12.38
416	s150×150×4	90000	273612	9.29
417	s150×150×5	112500	342015	12.07
418	s150×150×6	135000	410418	14.85

Table A.3 (cont.): 3D solid element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
419	f5×5×75	7200	16200	0.31
420	f5×5×100	9600	21600	0.42
421	f5×5×125	12000	27000	0.53
422	f5×5×150	14400	32400	0.63
423	f5×5×175	16800	37800	0.74
424	f5×5×200	19200	43200	0.84
425	f5×5×250	24000	54000	1.05
426	f5×10×75	13575	29700	0.59
427	f5×10×100	18100	39600	0.79
428	f5×10×125	22625	49500	0.99
429	f5×10×150	27150	59400	1.18
430	f5×10×175	31675	69300	1.38
431	f5×10×200	36200	79200	1.58
432	f5×10×250	45250	99000	1.97
433	f5×15×75	19950	43200	0.87
434	f5×15×100	26600	57600	1.16
435	f5×15×125	33250	72000	1.45
436	f5×15×150	39900	86400	1.74
437	f5×15×175	46550	100800	2.03
438	f5×15×200	53200	115200	2.31
439	f5×15×250	66500	144000	2.89
440	f5×20×75	26325	56700	1.14
441	f5×20×100	35100	75600	1.52
442	f5×20×125	43875	94500	1.91
443	f5×20×150	52650	113400	2.29
444	f5×20×175	61425	132300	2.67
445	f5×20×200	70200	151200	3.05
446	f5×20×250	87750	189000	3.82
447	f10×10×75	25575	54450	1.11
448	f10×10×100	34100	72600	1.48
449	f10×10×125	42625	90750	1.85
450	f10×10×150	51150	108900	2.22
451	f10×10×175	59675	127050	2.59
452	f10×10×200	68200	145200	2.96
453	f10×10×250	85250	181500	3.7
454	f10×15×75	37575	79200	1.62
455	f10×15×100	50100	105600	2.17
456	f10×15×125	62625	132000	2.71
457	f10×15×150	75150	158400	3.25
458	f10×15×175	87675	184800	3.8
459	f10×15×200	100200	211200	4.34
460	f10×15×250	125250	264000	5.43

Table A.4: 3D frame element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
461	f10×20×75	49575	103950	2.14
462	f10×20×100	66100	138600	2.86
463	f10×20×125	82625	173250	3.57
464	f10×20×150	99150	207900	4.29
465	f10×20×175	115675	242550	5
466	f10×20×200	132200	277200	5.72
467	f10×20×250	165250	346500	7.15
468	f15×15×75	55200	115200	2.38
469	f15×15×100	73600	153600	3.18
470	f15×15×125	92000	192000	3.97
471	f15×15×150	110400	230400	4.77
472	f15×15×175	128800	268800	5.57
473	f15×15×200	147200	307200	6.37
474	f15×15×250	184000	384000	7.96
475	f15×20×75	72825	151200	3.14
476	f15×20×100	97100	201600	4.19
477	f15×20×125	121375	252000	5.24
478	f15×20×150	145650	302400	6.29
479	f15×20×175	169925	352800	7.34
480	f15×20×200	194200	403200	8.39
481	f15×20×250	242750	504000	10.49
482	f20×20×75	96075	198450	4.14
483	f20×20×100	128100	264600	5.52
484	f20×20×125	160125	330750	6.91
485	f20×20×150	192150	396900	8.29
486	f20×20×175	224175	463050	9.68
487	f20×20×200	256200	529200	11.06
488	f20×20×250	320250	661500	13.83
489	f10×10×10	3410	7260	0.14
490	f10×10×15	5115	10890	0.22
491	f10×10×20	6820	14520	0.29
492	f10×10×25	8525	18150	0.37
493	f10×10×30	10230	21780	0.44
494	f10×10×35	11935	25410	0.51
495	f10×10×40	13640	29040	0.59
496	f10×10×45	15345	32670	0.66
497	f10×10×50	17050	36300	0.74

Table A.4 (cont.): 3D frame element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
498	f10×15×15	7515	15840	0.32
499	f10×15×20	10020	21120	0.43
500	f10×15×25	12525	26400	0.54
501	f10×15×30	15030	31680	0.65
502	f10×15×35	17535	36960	0.75
503	f10×15×40	20040	42240	0.86
504	f10×15×45	22545	47520	0.97
505	f10×15×50	25050	52800	1.08
506	f10×20×20	13220	27720	0.56
507	f10×20×25	16525	34650	0.71
508	f10×20×30	19830	41580	0.85
509	f10×20×35	23135	48510	0.99
510	f10×20×40	26440	55440	1.14
511	f10×20×45	29745	62370	1.28
512	f10×20×50	33050	69300	1.42
513	f10×25×25	20525	42900	0.88
514	f10×25×30	24630	51480	1.06
515	f10×25×35	28735	60060	1.23
516	f10×25×40	32840	68640	1.41
517	f10×25×45	36945	77220	1.59
518	f10×25×50	41050	85800	1.77
519	f10×30×30	29430	61380	1.26
520	f10×30×35	34335	71610	1.47
521	f10×30×40	39240	81840	1.69
522	f10×30×45	44145	92070	1.9
523	f10×30×50	49050	102300	2.11
524	f10×35×35	39935	83160	1.71
525	f10×35×40	45640	95040	1.96
526	f10×35×45	51345	106920	2.21
527	f10×35×50	57050	118800	2.46
528	f15×15×15	11040	23040	0.47
529	f15×15×20	14720	30720	0.63
530	f15×15×25	18400	38400	0.79
531	f15×15×30	22080	46080	0.95
532	f15×15×35	25760	53760	1.11
533	f15×15×40	29440	61440	1.27
534	f15×15×45	33120	69120	1.43
535	f15×15×50	36800	76800	1.58

Table A.4 (cont.): 3D frame element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
536	f15×20×20	19420	40320	0.83
537	f15×20×25	24275	50400	1.04
538	f15×20×30	29130	60480	1.25
539	f15×20×35	33985	70560	1.46
540	f15×20×40	38840	80640	1.67
541	f15×20×45	43695	90720	1.88
542	f15×20×50	48550	100800	2.09
543	f15×25×25	30150	62400	1.29
544	f15×25×30	36180	74880	1.55
545	f15×25×35	42210	87360	1.81
546	f15×25×40	48240	99840	2.07
547	f15×25×45	54270	112320	2.33
548	f15×25×50	60300	124800	2.59
549	f15×30×30	43230	89280	1.85
550	f15×30×35	50435	104160	2.16
551	f15×30×40	57640	119040	2.47
552	f15×30×45	64845	133920	2.79
553	f15×30×50	72050	148800	3.1
554	f15×35×35	58660	120960	2.51
555	f15×35×40	67040	138240	2.88
556	f15×35×45	75420	155520	3.24
557	f15×35×50	83800	172800	3.6
558	f20×20×20	25620	52920	1.09
559	f20×20×25	32025	66150	1.37
560	f20×20×30	38430	79380	1.65
561	f20×20×35	44835	92610	1.92
562	f20×20×40	51240	105840	2.2
563	f20×20×45	57645	119070	2.48
564	f20×20×50	64050	132300	2.75
565	f20×25×25	39775	81900	1.7
566	f20×25×30	47730	98280	2.04
567	f20×25×35	55685	114660	2.39
568	f20×25×40	63640	131040	2.73
569	f20×25×45	71595	147420	3.07
570	f20×25×50	79550	163800	3.42
571	f20×30×30	57030	117180	2.44
572	f20×30×35	66535	136710	2.85
573	f20×30×40	76040	156240	3.26
574	f20×30×45	85545	175770	3.67
575	f20×30×50	95050	195300	4.08
576	f20×35×35	77385	158760	3.31
577	f20×35×40	88440	181440	3.79
578	f20×35×45	99495	204120	4.27
579	f20×35×50	110550	226800	4.75

Table A.4 (cont.): 3D frame element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
580	f25×25×25	49400	101400	2.11
581	f25×25×30	59280	121680	2.54
582	f25×25×35	69160	141960	2.96
583	f25×25×40	79040	162240	3.39
584	f25×25×45	88920	182520	3.82
585	f25×25×50	98800	202800	4.24
586	f25×30×30	70830	145080	3.03
587	f25×30×35	82635	169260	3.54
588	f25×30×40	94440	193440	4.05
589	f25×30×45	106245	217620	4.56
590	f25×30×50	118050	241800	5.07
591	f25×35×35	96110	196560	4.11
592	f25×35×40	109840	224640	4.71
593	f25×35×45	123570	252720	5.3
594	f25×35×50	137300	280800	5.89
595	f30×30×30	84630	172980	3.62
596	f30×30×35	98735	201810	4.23
597	f30×30×40	112840	230640	4.83
598	f30×30×45	126945	259470	5.44
599	f30×30×50	141050	288300	6.05
600	f30×35×35	114835	234360	4.91
601	f30×35×40	131240	267840	5.62
602	f30×35×45	147645	301320	6.33
603	f30×35×50	164050	334800	7.04
604	f35×35×35	133560	272160	5.71
605	f35×35×40	152640	311040	6.54
606	f35×35×45	171720	349920	7.36
607	f35×35×50	190800	388800	8.18
608	f25×25×4	7904	16224	0.32
609	f25×25×5	9880	20280	0.4
610	f25×25×6	11856	24336	0.49
611	f25×50×4	15604	31824	0.63
612	f25×50×5	19505	39780	0.79
613	f25×50×6	23406	47736	0.96
614	f25×75×4	23304	47424	0.93
615	f25×75×5	29130	59280	1.19
616	f25×75×6	34956	71136	1.44
617	f25×100×4	31004	63024	1.24
618	f25×100×5	38755	78780	1.58
619	f25×100×6	46506	94536	1.91
620	f25×125×4	38704	78624	1.55
621	f25×125×5	48380	98280	1.97
622	f25×125×6	58056	117936	2.38
623	f25×150×4	46404	94224	1.86
624	f25×150×5	58005	117780	2.36
625	f25×150×6	69606	141336	2.86

Table A.4 (cont.): 3D frame element models with regular geometry

Model No.	Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
626	f50×50×4	30804	62424	1.23
627	f50×50×5	38505	78030	1.57
628	f50×50×6	46206	93636	1.9
629	f50×75×4	46004	93024	1.84
630	f50×75×5	57505	116280	2.34
631	f50×75×6	69006	139536	2.83
632	f50×100×4	61204	123624	2.45
633	f50×100×5	76505	154530	3.11
634	f50×100×6	91806	185436	3.77
635	f50×125×4	76404	154224	3.06
636	f50×125×5	95505	192780	3.88
637	f50×125×6	114606	231336	4.7
638	f50×150×4	91604	184824	3.67
639	f50×150×5	114505	231030	4.65
640	f50×150×6	137406	277236	5.64
641	f75×75×4	68704	138624	2.75
642	f75×75×5	85880	173280	3.49
643	f75×75×6	103056	207936	4.23
644	f75×100×4	91404	184224	3.66
645	f75×100×5	114255	230280	4.64
646	f75×100×6	137106	276336	5.63
647	f75×125×4	114104	229824	4.57
648	f75×125×5	142630	287280	5.8
649	f75×125×6	171156	344736	7.02
650	f75×150×4	136804	275424	5.48
651	f75×150×5	171005	344280	6.95
652	f75×150×6	205206	413136	8.42
653	f100×100×4	121604	244824	4.87
654	f100×100×5	152005	306030	6.18
655	f100×100×6	182406	367236	7.48
656	f100×125×4	151804	305424	6.08
657	f100×125×5	189755	381780	7.71
658	f100×125×6	227706	458136	9.34
659	f100×150×4	182004	366024	7.28
660	f100×150×5	227505	457530	9.24
661	f100×150×6	273006	549036	11.2
662	f125×125×4	189504	381024	7.58
663	f125×125×5	236880	476280	9.62
664	f125×125×6	284256	571536	11.66
665	f125×150×4	227204	456624	9.09
666	f125×150×5	284005	570780	11.54
667	f125×150×6	340806	684936	13.98
668	f150×150×4	272404	547224	10.9
669	f150×150×5	340505	684030	13.83
670	f150×150×6	408606	820836	16.76

Table A.4 (cont.): 3D frame element models with regular geometry

A.2 Test Problems with Irregular Geometry

Figure A.6 shows selected 2D test problems with irregular geometries. FE Models shown in Figure A.6 are for quadrilateral elements. FE Models with frame elements have the same geometry. Figure A.7 shows selected 3D test problems with irregular geometries. FE Models shown in Figure A.7 are for solid elements. FE Models with 3D frame elements have the same geometry.

Figure A.8 shows the non-zero pattern for the test problem q-varying-4. Similarly, Figure A.9 shows the non-zero pattern for the test problem s-bldg58.

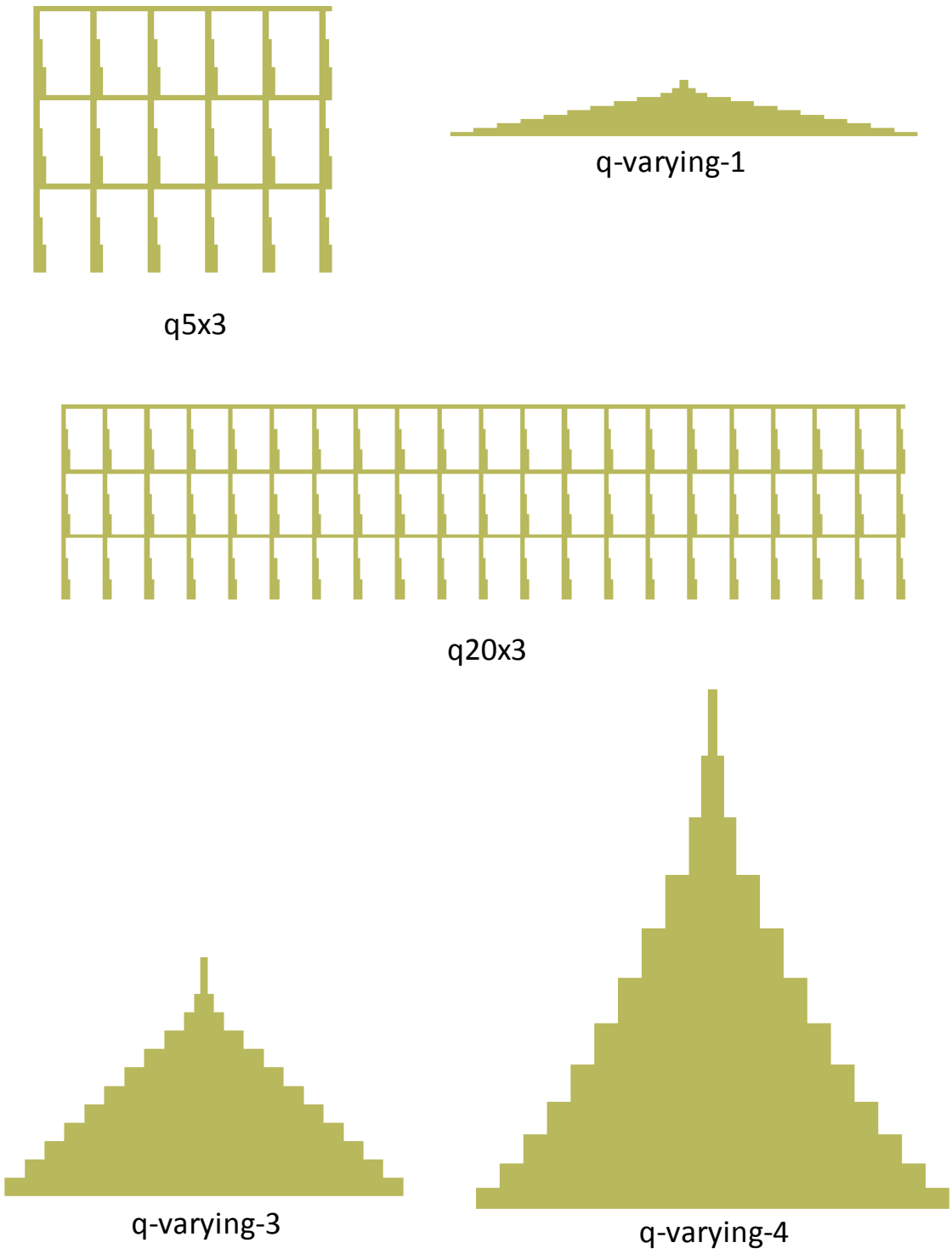


Figure A.6: Selected 2D test problems with quadrilateral elements.

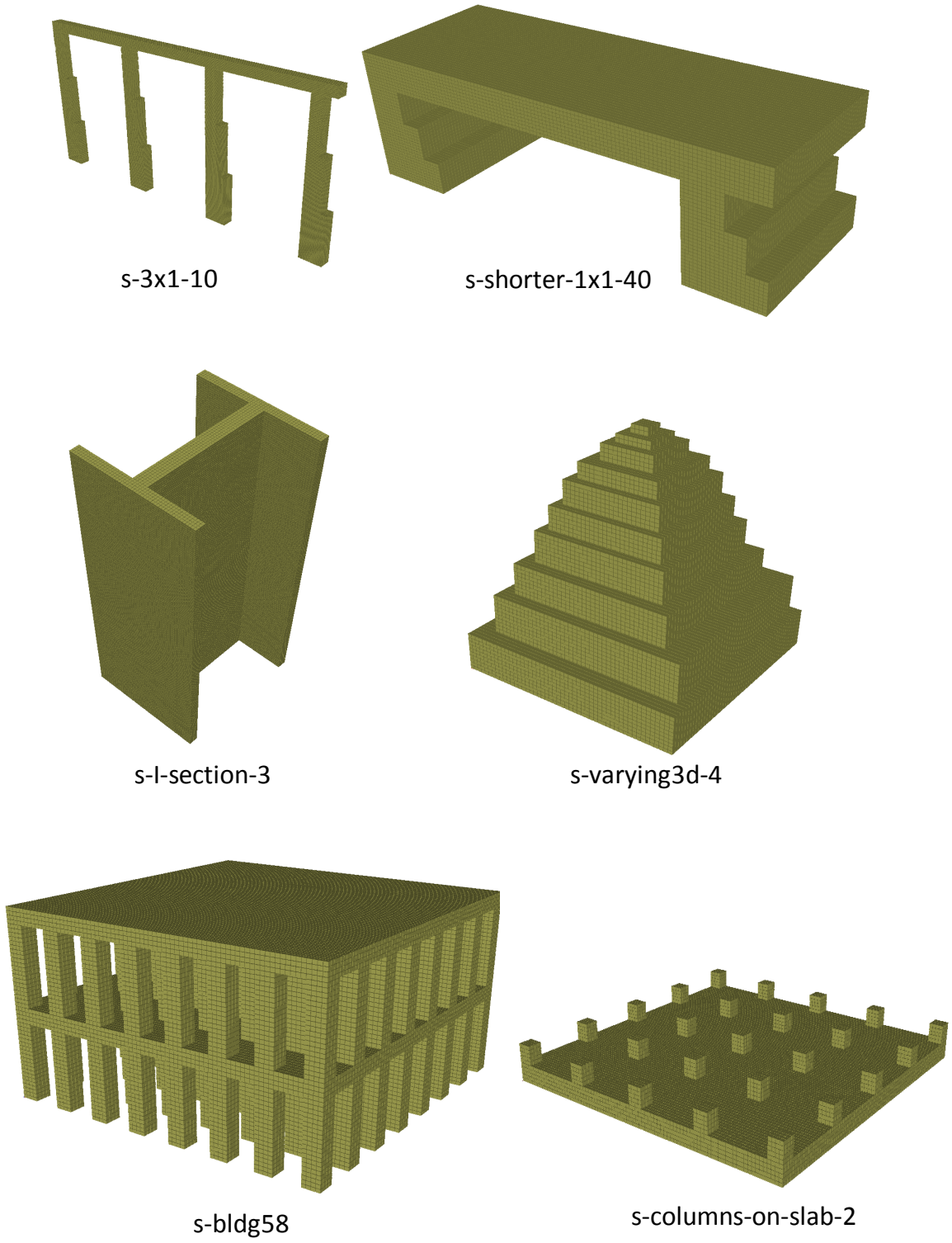


Figure A.7: Selected 3D test problems with solid elements.

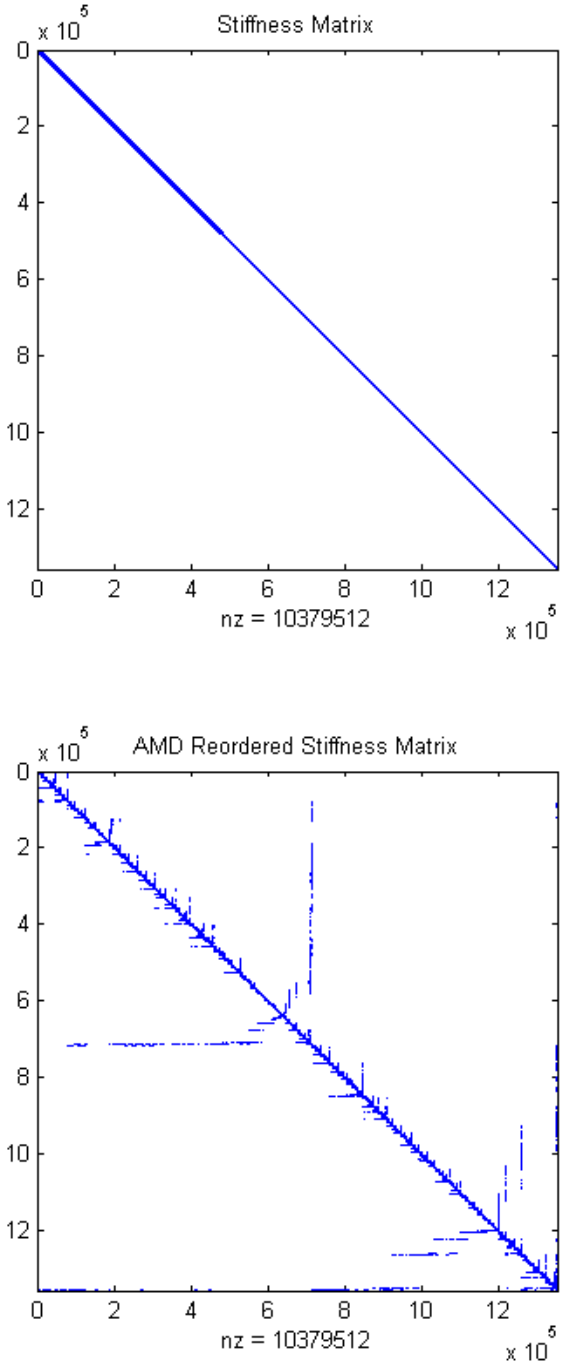


Figure A.8: Non-zero pattern for the test problem q-varying-4 (original ordering and AMD matrix ordering).

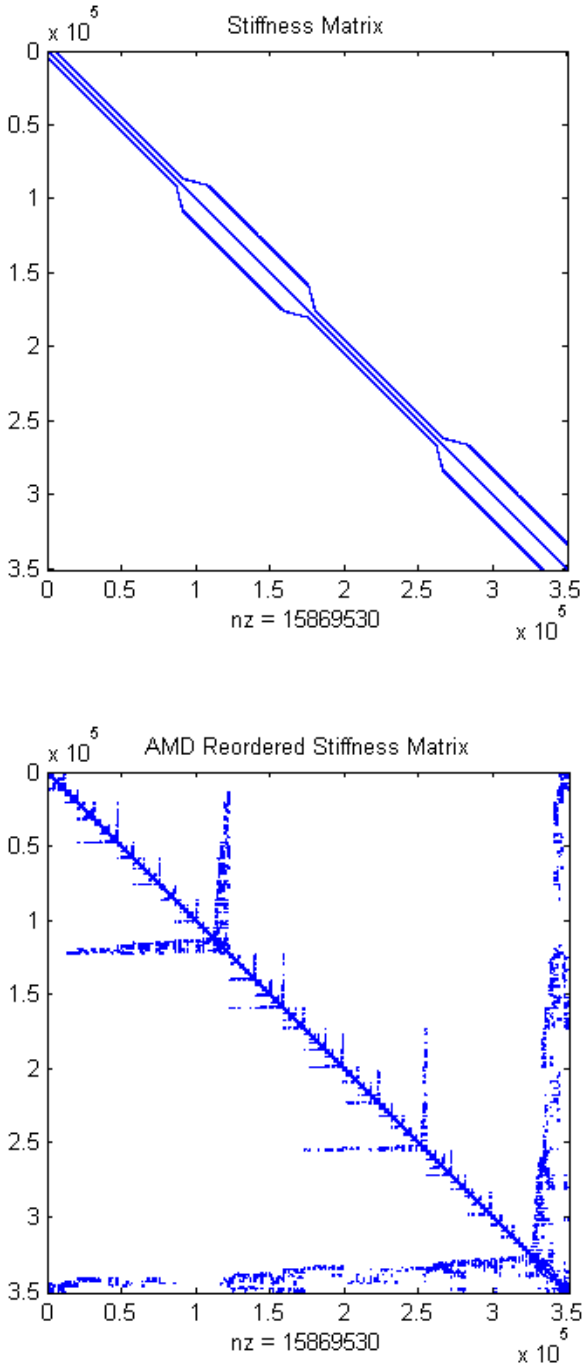


Figure A.9: Non-zero pattern for the test problem s-bldg58 (original ordering and AMD matrix ordering).

Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
q5×3	54600	117090	1.06
q5×5	91000	195150	1.77
q5×10	182000	390300	3.55
q5×15	273000	585450	5.32
q5×20	364000	780600	7.1
q10×3	101850	218460	1.98
q10×5	169750	364100	3.31
q10×10	339500	728200	6.62
q10×15	509250	1092300	9.93
q10×20	679000	1456400	13.24
q20×3	196350	421200	3.82
q20×5	327250	702000	6.38
q20×10	654500	1404000	12.76
q20×15	981750	2106000	19.14
q20×20	1309000	2808000	25.52
q-varying-1	55900	112060	1.05
q-varying-2	229550	459980	4.35
q-varying-3	279500	560300	5.3
q-varying-4	450700	903920	8.56
q-varying-5	987500	1980700	18.77
q-varying-6	1975000	3961400	37.55

Table A.5: 2D quadrilateral element models with irregular geometry

Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
f5×3	111930	175635	1.34
f5×5	186550	292725	2.23
f5×10	373100	585450	4.47
f5×15	559650	878175	6.7
f5×20	746200	1170900	8.93
f10×3	208680	327690	2.5
f10×5	347800	546150	4.16
f10×10	695600	1092300	8.32
f10×15	1043400	1638450	12.49
f10×20	1391200	2184600	16.65
f20×3	402180	631800	4.81
f20×5	670300	1053000	8.02
f20×10	1340600	2106000	16.03
f20×15	2010900	3159000	24.05
f20×20	2681200	4212000	32.07
f-varying-1	111930	168090	1.33
f-varying-2	459540	689970	5.51
f-varying-3	559650	840450	6.71
f-varying-4	902660	1355880	10.83
f-varying-5	1977850	2971050	23.73
f-varying-6	3955700	5942100	47.48

Table A.6: 2D frame element models with irregular geometry

Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
s-3×1-5	47600	191355	6.48
s-5×1-5	72800	292725	9.92
s-10×1-5	135800	546150	18.5
s-3×1-10	119000	420981	15.44
s-5×1-10	182000	643995	23.62
s-1×1-20	112000	377937	14.28
s-shorter-1×1-20	40000	136017	5.07
s-shorter-1×1-40	80000	265557	10.06
s-I-section-1	52500	191925	6.88
s-I-section-2	24500	89565	3.18
s-I-section-3	105000	383850	13.81
s-bldg58	84768	351150	11.64
s-bldg59	79488	307920	10.64
s-columns-on-slab-1	9056	31050	0.58
s-columns-on-slab-2	37280	117750	4.08
s-varying3d-1	16680	53100	1.89
s-varying3d-2	48600	157674	5.93
s-varying3d-3	58320	188520	7.15
s-varying3d-4	70920	224340	8.68
s-varying3d-5	38808	122598	4.65
s-varying3d-6	25944	83178	3.06
s-varying3d-7	11476	38985	1.26

Table A.7: 3D solid element models with irregular geometry

Model Name	No. of Elements	No. of Dofs	Non-zero in **K** (1e6)
f-3×1-5	172090	382710	7.45
f-5×1-5	262990	585450	11.38
f-10×1-5	490240	1092300	21.2
f-3×1-10	393610	841962	16.93
f-5×1-10	601510	1287990	25.86
f-1×1-20	359910	755874	15.46
f-shorter-1×1-20	126870	272034	5.38
f-shorter-1×1-40	249670	531114	10.58
f-I-section-1	182325	383850	7.81
f-I-section-2	85085	179130	3.63
f-I-section-3	364650	767700	15.64
f-bldg58	300200	702300	12.75
f-bldg59	257304	615840	10.6
f-columns-on-slab-1	29630	62100	1.02
f-columns-on-slab-2	115650	235500	4.73
f-varying3d-1	52060	106200	2.15
f-varying3d-2	153606	315348	6.54
f-varying3d-3	183880	377040	7.85
f-varying3d-4	220420	448680	9.41
f-varying3d-5	120506	245196	5.1
f-varying3d-6	81350	166356	3.42
f-varying3d-7	37359	77970	1.52

Table A.8: 3D frame element models with irregular geometry

APPENDIX B:

UTILITY PROGRAMS

In addition to the direct solver package SES, the following programs are developed throughout this study:

- **Utility Library**

The utility library provides abstractions for dense matrices (packed and full), handling sparse indices, multithreading, assembly on dense matrices, and partial factorization operations. The MKL library is linked to the utility library. Therefore, there is no connection between the direct solver package and the BLAS libraries. The direct solver package calls the factorization subroutines in the utility libraries, which makes the BLAS library calls (MKL).

- **Input Generator**

An input generator is developed to easily create test problems with different geometries. The input generator creates models with unit size elements. The input generator can be used to create models with 2D quadrilateral, 2D frame, 3D solid and 3D frame elements. Models with regular prismatic geometries can be created easily using command line arguments. For irregular geometries, an input file is required. The output of the input generator is an input file for the direct solver package, which contains element stiffness matrices, element connectivity information, node coordinates and support conditions.

- **Input Converter**

The input converter converts a SES input file to the matrix market coordinate sparse matrix format [156].

- **SES Viewer**

SES Viewer is a 3D visualization tool for the test problems and results of the preprocessing algorithms. It employs OpenSceneGraph library [157] for 3D graphics. All figures illustrating the geometry of the test problems are produced using SES Viewer. Furthermore, any pivot-ordering found in the preprocessing phase can be monitored in a step-by-step fashion using the 3D GUI of the SES Viewer. Figure B.1 shows a screenshot from SES Viewer.

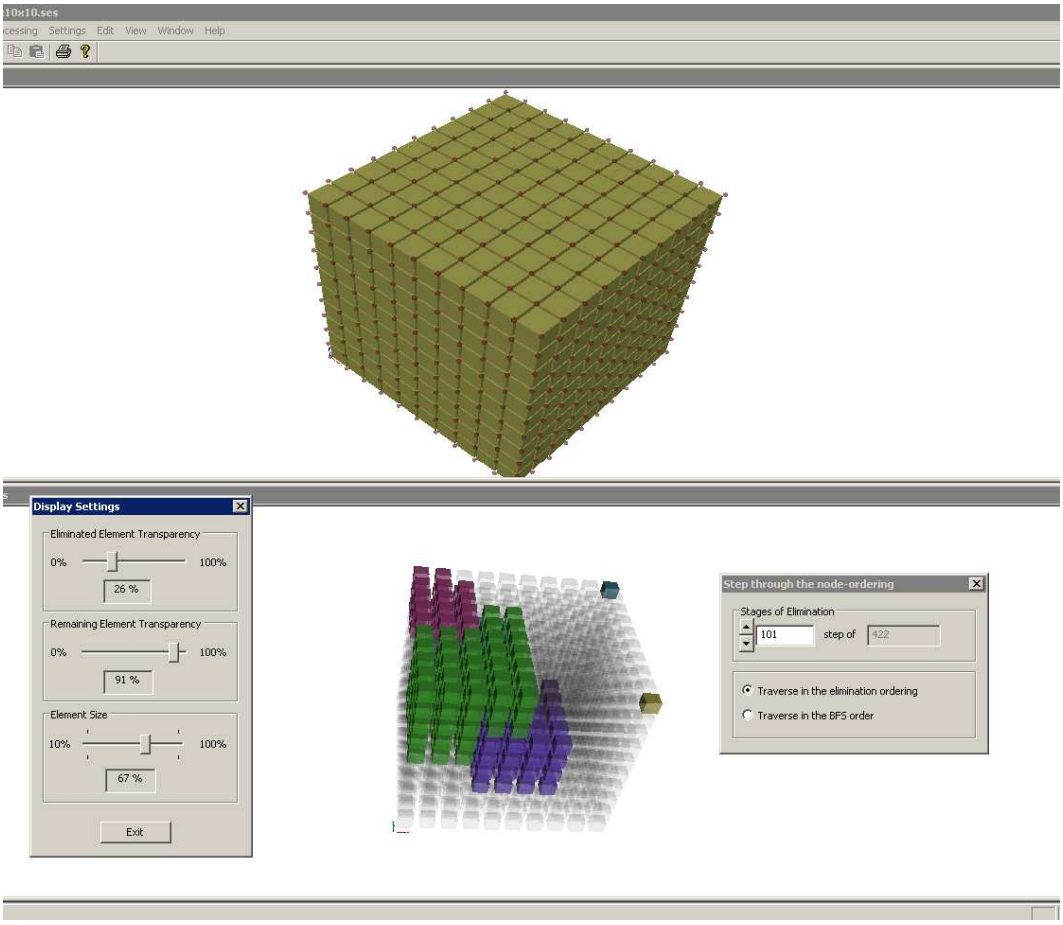


Figure B.1: A screenshot from SES Viewer. A 3D cubic model is at the top and the monitoring of a pivot-ordering for the model is at the bottom.

- **BLAS/LAPACK Performance Evaluation**

The performance of BLAS and LAPACK subroutines in the MKL library can be evaluated using the performance evaluation program. This program executes the desired BLAS/LAPACK subroutine repeatedly inside a loop. The execution time of the subroutine is recorded. The program gives the speed of a BLAS/LAPACK by dividing the operation count required for the subroutine by the execution time.

- **Partial Factorization Simulator**

This program takes an assembly tree and simulates the partial factorization operations using corresponding BLAS/LAPACK subroutines. It gives the partial factorization time for an assembly tree. It also gives the operation counts for the partial factorization and average partial factorization speeds.

REFERENCES

[1] AMD, "Multicore Processing: Next Evolution in Computing (AMD White Papers)," 2005.

[2] J. Held, J. Bautista, and S. Koehl, "From a Few Cores to Many: A Tera-scale Computing Research Overview (Intel White Paper)," Intel Corporation, 2006.

[3] Microsoft, "The Manycore Shift, Microsoft Parallel Computing Initiative Ushers Computing into the Next Era (Microsoft White Paper)," Available: http://www.microsoft.com/downloads/details.aspx?FamilyId=633F9F08-AAD9-46C4-8CAE-B204472838E1&displaylang=en, (Accessed: April 10, 2010).

[4] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," University of Virginia1994.

[5] J. L. Hennessy and D. A. Patterson, Computer Architecture, A Quantitative Approach, 4 ed.: Morgan Kaufmann, 2006.

[6] Intel, "Math Kernel Library (MKL), " Available: http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm, (Accessed: April 10, 2010)

[7] AMD, "AMD Core Math Library (ACML), " Available: http://developer.amd.com/cpu/libraries/acml/Pages/default.aspx, (Accessed: April 10, 2010).

[8] IBM, "Engineering Scientific Subroutine Library (ESSL), " Available: http://www-03.ibm.com/systems/software/essl/index.html, (Accessed: April 10, 2010).

[9] "LAPACK: Linear Algebra Package, " Available: http://www.netlib.org/lapack/, (Accessed: April 10, 2010).

[10] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," ACM Trans. Math. Softw., vol. 16, pp. 1-17, 1990.

[11] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," ACM Trans. Math. Softw., vol. 14, pp. 1-17, 1988.

[12] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," ACM Trans. Math. Softw., vol. 5, pp. 308-323, 1979.

[13] K. Goto and R. v. d. Geijn, "On Reducing TLB Misses in Matrix Multiplication," University of Texas at Austin2002.

[14] K. Goto and R. V. D. Geijn, "High-performance Implementation of the Level-3 BLAS," ACM Trans. Math. Softw., vol. 35, pp. 1-14, 2008.

[15] "Automatically Tuned Linear Algebra Software (ATLAS), " Available: http://math-atlas.sourceforge.net/, (Accessed: April 10, 2010)

[16] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated Empirical Optimization of Software and the ATLAS Project," Parallel Computing, vol. 27, pp. 3-35, 2001.

[17] I. S. Duff, "The Impact of High-performance Computing in the Solution of Linear Systems: Trends and Problems," J. Comput. Appl. Math., vol. 123, pp. 515-530, 2000.

[18] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. v. d. Vorst, Numerical Linear Algebra for High Performance Computers: Society for Industrial and Applied Mathematics, 1998.

[19] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," Technical Report No. UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley2006.

[20] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A Library of Automatically Tuned Sparse Matrix Kernels," Journal of Physics: Conference Series, vol. 16, pp. 521-530, 2005.

[21] R. W. Vuduc, "Automatic Performance Tuning of Sparse Matrix Kernels," University of California, Berkeley, 2003.

[22] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms," Parallel Comput., vol. 35, pp. 178-194, 2009.

[23] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," Technical Report #1593, University of Wisconsin, Madison2007.

[24] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The Impact of Performance Asymmetry in Emerging Multicore Architectures," SIGARCH Comput. Archit. News, vol. 33, pp. 506-517, 2005.

[25] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," presented at the Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, 2003.

[26] T. R. Halfhill. (2008) Parallel Processing with CUDA. Microprocessor Report

[27] "NVidia CUDA Zone, " Available: http://www.nvidia.com/object/cuda_home.html#, (Accessed: April 10, 2010).

[28] "OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems, " Available: http://www.khronos.org/opencl/, (Accessed: April 10, 2010).

[29] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel Computing Experiences with CUDA," IEEE Micro, vol. 28, pp. 13-27, 2008.

[30] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures," Parallel Computing, vol. 35, pp. 38-53, 2009.

[31] "The Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) Project, " Available: http://icl.cs.utk.edu/plasma/index.html, (Accessed: April 10, 2010).

[32] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra, "A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators," Technical Report, Innovative Computing Laboratory, University of Tennessee2009.

[33] "Matrix Algebra on GPU and Multicore Architectures (MAGMA) Project, " Available: http://icl.cs.utk.edu/magma/index.html, (Accessed: April 10, 2010).

[34] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors," presented at the Proceedings of the 34th Annual International Symposium on Computer Architecture, San Diego, California, USA, 2007.

[35] T. A. Davis, Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2): Society for Industrial and Applied Mathematics, 2006.

[36] I. S. Duff, A. M. Erisman, and J. K. Reid, Direct Methods for Sparse matrices: Oxford University Press, Inc., 1986.

[37] J. A. Scott and Y. Hu, "Experiences of Sparse Direct Symmetric Solvers," ACM Trans. Math. Softw., vol. 33, p. 18, 2007.

[38] E. Rozin and S. Toledo, "Locality of Reference in Sparse Cholesky Factorization Methods," Electronic Transactions on Numerical Analysis, vol. 21, pp. 81-106, 2005.

[39] E. G. Ng and B. W. Peyton, "Block Sparse Cholesky Algorithms on Advanced Uniprocessor Computers," SIAM Journal on Scientific Computing, vol. 14, pp. 1034-1056, 1993.

[40] E. Rothberg and A. Gupta, "An Evaluation of Left-Looking, Right-Looking and Multifrontal Approaches to Sparse Cholesky Factorization on Hierarchical-Memory Machines," Technical Report STAN-CS-91-1377, Department of Computer Science, Stanford University1991.

[41] E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," presented at the Proceedings of the 1969 24th National Conference, 1969.

[42] L. Wai-Hung and H. S. Andrew, "Comparative Analysis of the Cuthill--McKee and the Reverse Cuthill--McKee Ordering Algorithms for Sparse Matrices," SIAM Journal on Numerical Analysis, vol. 13, pp. 198-213, 1976.

[43] N. E. Gibbs, W. G. J. Poole, and P. K. Stockmeyer "An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix.," SIAM J. Numer. Anal. , vol. 18, pp. 235-251, 1976.

[44] S. W. Sloan, "An Algorithm for Profile and Wavefront Reduction of Sparse Matrices," International Journal for Numerical Methods in Engineering, vol. 23, pp. 239-251, 1986.

[45] B. M. Irons, "A Frontal Solution Scheme for Finite Element Analysis," Int. J. Numer. Methods Eng. , vol. 2, pp. 5-32, 1970.

[46] J. A. Scott, "On Ordering Elements for a Frontal Solver," Communications in Numerical Methods in Engineering, vol. 15, pp. 309-323, 1999.

[47] A. Bykat, "A Note on an Element Ordering Scheme," International Journal for Numerical Methods in Engineering, vol. 11, pp. 194-198, 1977.

[48] H. L. Pina, "An Algorithm for Frontwidth Reduction," International Journal for Numerical Methods in Engineering, vol. 17, pp. 1539-1547, 1981.

[49] A. Razzaque, "Automatic Reduction of Frontwidth for Finite Element Analysis," International Journal for Numerical Methods in Engineering, vol. 15, pp. 1315-1324, 1980.

[50] S. W. Sloan and M. F. Randolph, "Automatic Element Reordering for Finite Element Analysis with Frontal Solution Schemes," International Journal for Numerical Methods in Engineering, vol. 19, pp. 1153-1181, 1983.

[51] I. S. Duff and J. K. Reid, "The Multifrontal Solution of Indefinite Sparse Symmetric Linear," ACM Trans. Math. Softw., vol. 9, pp. 302-325, 1983.

[52] J. W. H. Liu, "The Multifrontal Method for Sparse Matrix Solution: Theory and Practice," SIAM Rev., vol. 34, pp. 82-109, 1992.

[53] E. Rothberg and R. Schreiber, "Efficient Methods for Out-of-Core Sparse Cholesky Factorization," SIAM J. Sci. Comput., vol. 21, pp. 129-144, 1999.

[54] J. W. H. Liu, "On the Storage Requirement in the Out-of-core Multifrontal Method for Sparse Factorization," ACM Trans. Math. Softw., vol. 12, pp. 249-264, 1986.

[55] A. Guermouche and J.-Y. L'excellent, "Constructing Memory-minimizing Schedules for Multifrontal Methods," ACM Trans. Math. Softw., vol. 32, pp. 17-32, 2006.

[56] J. W. H. Liu, "Equivalent Sparse Matrix Reordering by Elimination Tree Rotations," SIAM J. Sci. Stat. Comput., vol. 9, pp. 424-444, 1988.

[57] C. Ashcraft and R. Grimes, "The Influence of Relaxed Supernode Partitions on the Multifrontal Method," ACM Trans. Math. Softw., vol. 15, pp. 291-309, 1989.

[58] I. S. Duff and J. A. Scott, "Towards an Automatic Ordering for a Symmetric Sparse Direct Solver," Rutherford Appleton Laboratory RAL-TR-2006-001, 2005.

[59] N. I. M. Gould and J. A. Scott, "A Numerical Evaluation of HSL Packages for the Direct Solution of Large Sparse, Symmetric Linear Systems of Equations," ACM Trans. Math. Softw., vol. 30, pp. 300-325, 2004.

[60] A. George and W. H. Liu, "The Evolution of the Minimum Degree Ordering Algorithm," SIAM Rev., vol. 31, pp. 1-19, 1989.

[61] J. W. H. Liu, "Modification of the Minimum-degree Algorithm by Multiple Elimination," ACM Trans. Math. Softw., vol. 11, pp. 141-153, 1985.

[62] I. A. Cavers, "Using Deficiency Measure For Tiebreaking the Minimum Degree Algorithm," University of British Columbia1989.

[63] P. R. Amestoy, T. A. Davis, and I. S. Duff, "An Approximate Minimum Degree Ordering Algorithm," SIAM J. Matrix Anal. Appl., vol. 17, pp. 886-905, 1996.

[64] E. Rothberg and S. C. Eisenstat, "Node Selection Strategies for Bottom-Up Sparse Matrix Ordering," SIAM J. Matrix Anal. Appl., vol. 19, pp. 682-695, 1998.

[65] E. G. Ng and P. Raghavan, "Performance of Greedy Ordering Heuristics for Sparse Cholesky Factorization," SIAM J. Matrix Anal. Appl., vol. 20, pp. 902-914, 1999.

[66] J. A. George, "Nested Dissection of a Regular Finite Element Mesh," SIAM J. Numer. Anal., vol. 10, pp. 345-363, 1973.

[67] C. Ashcraft and J. W. H. Liu, "Robust Ordering of Sparse Matrices using Multisection," SIAM J. Matrix Anal. Appl., vol. 19, pp. 816-832, 1998.

[68] B. Hendrickson and E. Rothberg, "Improving the Run Time and Quality of Nested Dissection Ordering," SIAM J. Sci. Comput., vol. 20, pp. 468-489, 1998.

[69] G. Karypis and V. Kumar, "METIS – A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0," University of Minnesota1998.

[70] F. Pellegrini, J. Roman, and P. Amestoy, "Hybridizing Nested Dissection and Halo Approximate Minimum Degree for Efficient Sparse Matrix Ordering," presented at the Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, 1999.

[71] J. Schulze, "Towards a Tighter Coupling of Bottom-Up and Top-Down Sparse Matrix Ordering Methods," Bit Numerical Mathematics, vol. 41, pp. 800-841, 2001.

[72] E. G. Boman, U. V. Catalyurek, C. Chevalier, K. D. Devine, I. Safro, and M. M. Wolf, "Advances in Parallel Partitioning, Load Balancing and Matrix Ordering for Scientific Computing," Journal of Physics: Conference Series, vol. 180, p. 012008, 2009.

[73] J. W. H. Liu, "The Role of Elimination Trees in Sparse Factorization," SIAM J. Matrix Anal. Appl., vol. 11, pp. 134-172, 1990.

[74] M. T. Heath, E. Ng, and B. W. Peyton, "Parallel Algorithms for Sparse Linear Systems," SIAM Rev., vol. 33, pp. 420-460, 1991.

[75] W.-Y. Lin, "Finding Optimal Ordering of Sparse Matrices for Column-Oriented Parallel Cholesky Factorization," J. Supercomput., vol. 24, pp. 259-277, 2003.

[76] M. V. Padmini, B. B. Madan, and B. N. Jain, "Reordering for Parallelism," International Journal of Computer Mathematics, vol. 67, pp. 373 - 390, 1998.

[77] A. Guermouche, J.-Y. L'Excellent, and G. Utard, "Impact of Reordering on the Memory of a Multifrontal Solver," Parallel Comput., vol. 29, pp. 1191-1218, 2003.

[78] A. Gupta, G. Karypis, and V. Kumar, "Highly Scalable Parallel Algorithms for Sparse Matrix Factorization," IEEE Trans. Parallel Distrib. Syst., vol. 8, pp. 502-520, 1997.

[79] G. Karypis and V. Kumar, "Analysis of Multilevel Graph Partitioning," presented at the Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM), San Diego, California, United States, 1995.

[80] S. Y. Fialko, "The Nested Substructures Method for Solving Large Finite-element Systems as Applied to Thin-walled Shells with High Ribs," International applied mechanics, vol. 39, pp. 324-331 2003.

[81] O. Kurc, "A Substructure Based Parallel Solution Framework for Solving Linear Structural Systems with Multiple Loading Conditions," PhD, Civil and Environmental Engineering, Georgia Institute of Technology, Atlanta, 2005.

[82] S. Y. Synn and R. E. Fulton, "Practical Strategy for Concurrent Substructure Analysis," Computers & Structures vol. 54, pp. 939-944, 1995.

[83] C. Farhat and E. Wilson, "A New Finite Element Concurrent Computer Program Architecture," International Journal for Numerical Methods in Engineering, vol. 24, pp. 1771-1792, 1987.

[84] Y.-S. Yang and S. H. Hsieh, "Iterative Mesh Partitioning Optimization for Parallel Nonlinear Dynamic Finite Element Analysis with Direct Substructuring," Computational Mechanics, vol. 28, pp. 456-468, 2002.

[85] B. Hendrickson, "Load Balancing Fictions, Falsehoods and Fallacies," Applied Mathematical Modelling, pp. 99-108, 2000.

[86] B. Hendrickson and T. G. Kolda, "Graph Partitioning Models for Parallel Computing," Parallel Comput., vol. 26, pp. 1519-1534, 2000.

[87] B. Hendrickson and K. Devine, "Dynamic Load Balancing in Computational Mechanics," Computer Methods in Applied Mechanics and Engineering, vol. 184, pp. 485-500, 2000.

[88] C. Ashcraft and R. Grimes, "SPOOLES: An Object-oriented Sparse Matrix Library," presented at the Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing, 1999.

[89] C. Walshaw and M. Cross, "Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm," SIAM J. Sci. Comput., vol. 22, pp. 63-80, 2000.

[90] O. Kurc and K. M. Will, "An Iterative Parallel Workload Balancing Framework for Direct Condensation of Substructures," Computer Methods in Applied Mechanics and Engineering, vol. 196, pp. 2084–2096, 2007.

[91] G. Karypis, K. Schloegel, and V. Kumar, "PARMETIS: A Parallel Graph Partitioning and Sparse Matrix Ordering Library - Version 3.1," 2003.

[92] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 8xx: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate," University of Florida TR-2006-005, 2006.

[93] T. A. Davis. "CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/downdate, " Available: http://www.cise.ufl.edu/research/sparse/cholmod/, (Accessed: April 10, 2010).

[94] P. Henon, P. Ramet, and J. Roman, "PASTIX: A High-performance Parallel Direct Solver for Sparse Symmetric Positive Definite Systems," Parallel Comput., vol. 28, pp. 301-321, 2002.

[95] "PASTIX Parallel Sparse Matrix Package, " Available: http://dept-info.labri.fr/~ramet/pastix/, (Accessed: April 10, 2010).

[96] "PSPASES : Scalable Parallel Direct Solver Library for Sparse Symmetric Positive Definite Systems," Available: http://www-users.cs.umn.edu/~mjoshi/pspases/, (Accessed: April 10, 2010).

[97] O. Schenk and K. Gartner, "Two-level Dynamic Scheduling in PARDISO: Improved Scalability on Shared Memory Multiprocessing Systems," Parallel Comput., vol. 28, pp. 187-197, 2002.

[98] O. Schenk and K. Gärtner. "PARDISO Solver Project, " Available: http://www.pardiso-project.org/, (Accessed: April 10, 2010).

[99] A. Gupta and M. Joshi, "WSMP: A High-performance Shared- and Distributed-memory Parallel Sparse Linear Equation Solver," IBM Research Division RC 22038, 2001.

[100] A. Gupta. "WSMP: Watson Sparse Matrix Package (Version 8.10.15), " Available: http://www-users.cs.umn.edu/~agupta/wsmp.html, (Accessed: April 10, 2010).

[101] F. Dobrian, G. Kumfert, and A. Pothen, "The Design of Sparse Direct Solvers Using Object-oriented Techniques," Institute for Computer Applications in Science and Engineering (ICASE)1999.

[102] T. A. Davis, "Algorithm 832: UMFPACK V4.3---An Unsymmetric-pattern Multifrontal Method," ACM Trans. Math. Softw., vol. 30, pp. 196-199, 2004.

[103] T. A. Davis. "UMFPACK: Unsymmetric Multifrontal Sparse LU Factorization Package," Available: http://www.cise.ufl.edu/research/sparse/umfpack/, (Accessed: April 10, 2010).

[104] C. Ashcraft. "SPOOLES 2.2 : SParse Object Oriented Linear Equations Solver," Available: http://www.netlib.org/linalg/spooles/spooles.2.2.html, (Accessed: April 10, 2010).

[105] X. S. Li, "An Overview of SuperLU: Algorithms, Implementation, and User Interface," ACM Trans. Math. Softw., vol. 31, pp. 302-325, 2005.

[106] "SuperLU," Available: http://crd.lbl.gov/~xiaoye/SuperLU/, (Accessed: April 10, 2010)

[107] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, "A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling," SIAM J. Matrix Anal. Appl., vol. 23, pp. 15-41, 2001.

[108] "MUMPS : A Parallel Sparse Direct Solver, " Available: http://graal.ens-lyon.fr/MUMPS/, (Accessed: April 10, 2010).

[109] V. Rotkin and S. Toledo, "The Design and Implementation of a New Out-of-core Sparse Cholesky Factorization Method," ACM Trans. Math. Softw., vol. 30, pp. 19-46, 2004.

[110] S. Toledo, D. Chen, and V. Rotkin. "TAUCS: A Library of Sparse Linear Solvers "Available: http://www.tau.ac.il/~stoledo/taucs/, (Accessed: April 10, 2010).

[111] N. I. M. Gould, J. A. Scott, and Y. Hu, "A Numerical Evaluation of Sparse Direct Solvers for the Solution of Large Sparse Symmetric Linear Systems of Equations," ACM Trans. Math. Softw., vol. 33, p. 10, 2007.

[112] A. Gupta and Y. Muliadi, "An Experimental Comparison of some Direct Sparse Solver Packages," presented at the Proceedings of the 15th International Parallel \& Distributed Processing Symposium, 2001.

[113] P. R. Amestoy, I. S. Duff, J.-Y. L'excellent, and X. S. Li, "Analysis and Comparison of two General Sparse Solvers for Distributed Memory Computers," ACM Trans. Math. Softw., vol. 27, pp. 388-421, 2001.

[114] T. A. Davis. "The University of Florida Sparse Matrix Collection," Available: http://www.cise.ufl.edu/research/sparse/matrices/, (Access: April 10, 2010).

[115] A. George and J. Liu, "An Object-Oriented Approach to the Design of a User Interface for a Sparse Matrix Package," SIAM J. Matrix Anal. Appl., vol. 20, pp. 953-969, 1999.

[116] G. Kumfert and A. Pothen, "An Object-oriented Collection of Minimum Degree Algorithms: Design, Implementation, and Experiences," Institute for Computer Applications in Science and Engineering (ICASE)1999.

[117] M. Sala, K. S. Stanley, and M. A. Heroux, "On the Design of Interfaces to Sparse Direct Solvers," ACM Trans. Math. Softw., vol. 34, pp. 1-22, 2008.

[118] "AMESOS: Direct Sparse Solver Package", Available: http://trilinos.sandia.gov/packages/amesos/, (Accessed: April 10, 2010).

[119] P. Raghavan, "DSCPACK: Domain-Separator Codes for the Parallel Solution of Sparse Linear Systems," Tech. Rep. CSE-02-004, Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802-61062002.

[120] "ScaLAPACK (Scalable LAPACK) "Available: http://www.netlib.org/scalapack/, (Accessed: April 10, 2010).

[121] F. Pellegrini, "Scotch 5.1 User's Guide," Technical report, LaBRI2008.

[122] F. Pellegrini, "PT-SCOTCH 5.1 User's guide," Technical report, LaBRI2008.

[123] E. D. Dolan and J. J. Moré, " Benchmarking Optimization Software with Performance Profiles," Mathematical Programming, vol. 91, pp. 201-213, 2002.

[124] A. Tersteeg, M. Gillespie, and R. Evans, "The Three Stages of Preparation for Optimizing Parallel Software," 2009, Available: http://software.intel.com/en-us/articles/the-three-stages-of-preparation-for-optimizing-parallel-software, (Accessed: April 10, 2010).

[125] "AMD CodeAnalyst," Available: http://developer.amd.com/cpu/CodeAnalyst/codeanalystwindows/Pages/default.aspx, (Accessed: April 10, 2010).

[126] L.-Q. Lee, J. G. Siek, and A. Lumsdaine, "Generic Graph Algorithms for Sparse Matrix Ordering," presented at the Proceedings of the Third International Symposium on Computing in Object-Oriented Parallel Environments, 1999.

[127] A. George and J. W. H. Liu, "A Fast Implementation of the Minimum Degree Algorithm Using Quotient Graphs," ACM Trans. Math. Softw., vol. 6, pp. 337-358, 1980.

[128] J. A. Scott and J. K. Reid, "Ordering Symmetric Sparse Matrices for Small Profile and Wavefront," International Journal for Numerical Methods in Engineering, vol. 45, pp. 1737-1755, 1999.

[129] C. Ashcraft, "Compressed Graphs and the Minimum Degree Algorithm," SIAM J. Sci. Comput., vol. 16, pp. 1404-1411, 1995.

[130] A. Kaveh and G. R. Roosta, "Comparative Study of Finite Element Nodal Ordering Methods," Engineering Structures, vol. 20, pp. 88-96, 1998.

[131] I. F. M. M. Glaucio H. Paulino, Marcelo Gattass, and Subrata Mukherjee, "Node and Element Resequencing Using the Laplacian of a Finite Element Graph: Part II-Implementation and Numerical Results," International Journal for Numerical Methods in Engineering, vol. 37, pp. 1531-1555, 1994.

[132] S. J. Fenves and K. H. Law, "A Two-step Approach to Finite Element Ordering," International Journal for Numerical Methods in Engineering, vol. 19, pp. 891-911, 1983.

[133] B. H. V. Topping and P. Ivanyi, "Partitioning of Tall Buildings Using Bubble Graph Representation," vol. 15, pp. 178-183, 2001.

[134] M. E. Guney and K. M. Will, "Coarsening Finite Element Meshes for a Multifrontal Solver," presented at the Proceedings of the Sixth International Conference on Engineering Computational Technology Athens, Greece, 2008.

[135] Boost-1.42.0. "BOOST C++ Libraries," Available: http://www.boost.org/, (Accessed: May 3, 2010)

[136] B. Hendrickson and E. Rothberg, "Effective Sparse Matrix Ordering: Just Around the BEND " in 8th SIAM Conference Parallel Processing for Scientific Computing 1997.

[137] K. Peeters. "tree.hh: an STL-like C++ tree class," Available: http://tree.phi-sci.com/, (Accessed: April 10, 2010).

[138] J. R. Gilbert, E. G. Ng, and B. W. Peyton, "An Efficient Algorithm to Compute Row and Column Counts for Sparse Cholesky Factorization," SIAM J. Matrix Anal. Appl., vol. 15, pp. 1075-1091, 1994.

[139] J. K. Reid and J. A. Scott, "An Out-of-core Sparse Cholesky Solver," ACM Trans. Math. Softw., vol. 36, pp. 1-33, 2009.

[140] "Intel® Math Kernel Library for Windows OS, User's Guide." 2009, Available: http://www.intel.com/software/products/, (Accessed: April 10, 2010)

[141] I. S. Duff, "MA57---A Code for the Solution of Sparse Symmetric Definite and Indefinite Systems," ACM Trans. Math. Softw., vol. 30, pp. 118-144, 2004.

[142] P. R. Amestoy and I. S. Duff, "Memory Management Issues in Sparse Multifrontal Methods on Multiprocessors," The International journal of Supercomputer Applications vol. 7, pp. 64-82, 1993.

[143] P. R. Amestoy, I. S. Duff, and C. Vomel, "Task Scheduling in an Asynchronous Distributed Memory Multifrontal Solver," SIAM J. Matrix Anal. Appl., vol. 26, pp. 544-565, 2005.

[144] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, "Hybrid Scheduling for the Parallel Solution of Linear Systems," Parallel Comput., vol. 32, pp. 136-156, 2006.

[145] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, ScaLAPACK User's Guide: Society for Industrial and Applied Mathematics, 1997.

[146] J. D. Hogg, J. K. Reid, and J. A. Scott, "A DAG-based Sparse Cholesky Solver for Multicore Architectures," SFTC Rutherford Appleton Laboratory Harwell Science and Innovation Campus, 2009.

[147] G. A. Geist and E. Ng, "Task Scheduling for Parallel Sparse Cholesky Factorization," Int. J. Parallel Program., vol. 18, pp. 291-314, 1990.

[148] A. Pothen and C. Sun, "A Mapping Algorithm for Parallel Sparse Cholesky Factorization," SIAM Journal on Scientific Computing, vol. 14, pp. 1253-1257, 1993.

[149] J. J. Dongarra and V. Eijkhout, "Numerical Linear Algebra Algorithms and Software," J. Comput. Appl. Math., vol. 123, pp. 489-514, 2000.

[150] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman, "Tuning the Performance of I/O-intensive Parallel Applications," presented at the Proceedings of the Fourth Workshop on I/O in Parallel and Distributed Systems: Part of the Federated Computing Research Conference, Philadelphia, Pennsylvania, United States, 1996.

[151] Microsoft®. "Microsoft Developer Network (MSDN) Library, " Available: http://msdn.microsoft.com/en-us/library/default.aspx, (Accessed: April 10, 2010).

[152] NVIDIA, "CUDA CUBLAS Library," 2008, Available: http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf,

[153] NVIDIA®. "The NVIDIA® Tesla™ S1070 Computing System, " Available: http://www.nvidia.com/object/product_tesla_s1070_us.html, (Accessed: April 10, 2010).

[154] NVIDIA, " NVIDIA's Next Generation CUDA Compute Architecture: Fermi™, v1.1. Whitepaper," 2009, Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, (Accessed: April 10, 2010)

[155] D. A. Patterson., "The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges," 2009, Available: http://www.nvidia.com/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf, (Accessed: April 10, 2010)

[156] "Matrix Market: File formats," Available: http://math.nist.gov/MatrixMarket/formats.html, (Accessed: April 10, 2010).

[157] "OpenSceneGraph," Available: http://www.openscenegraph.org/projects/osg, (Accessed: April 20, 2010).